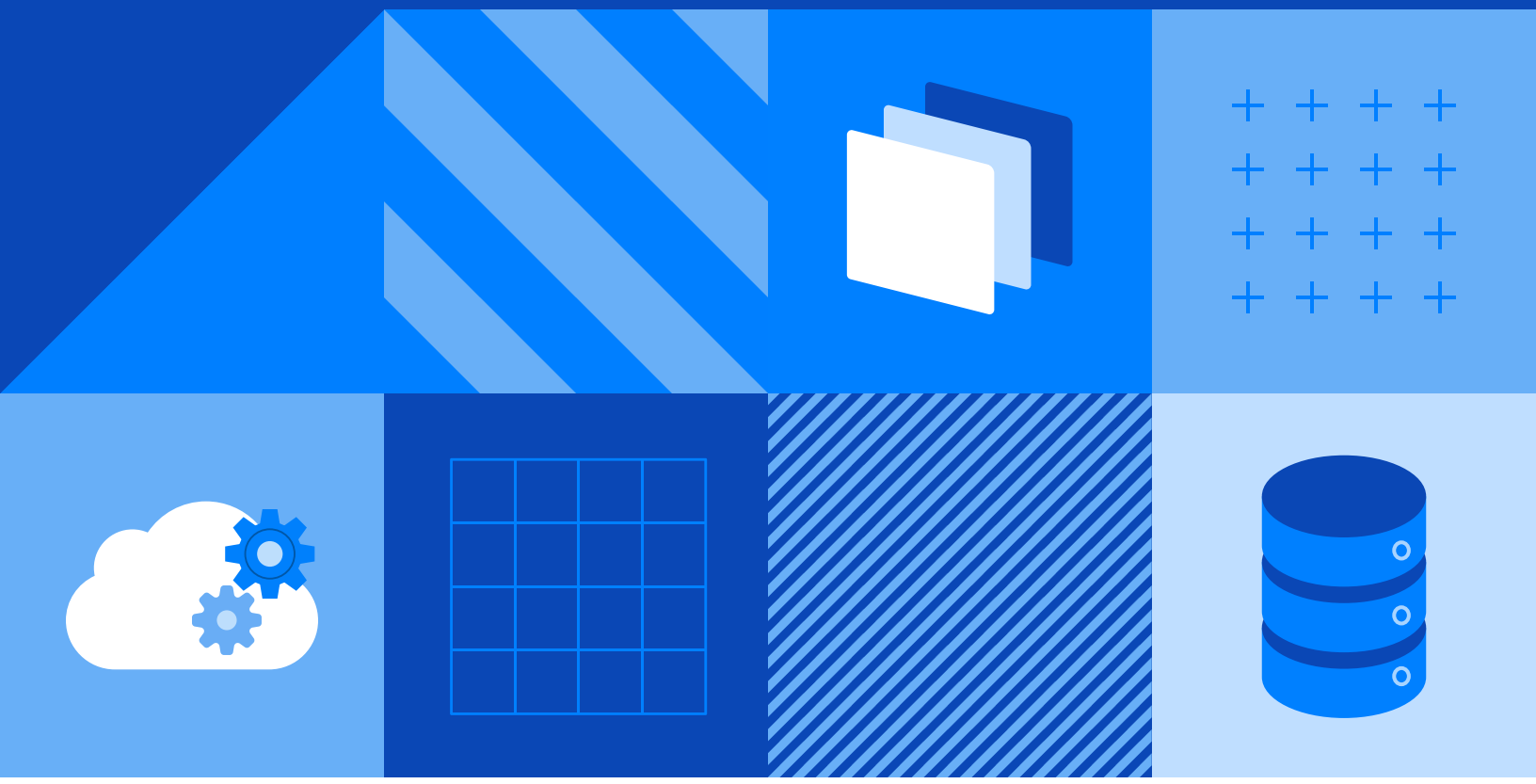
The Fivetran Protocol: How to Build a Bulletproof Data Replication API

*by **Meel Velliste, CTO at Fivetran***



Fivetran

If your company is like most SaaS application companies, a vast majority of your users want to upload the data from your application – along with the data from all the other SaaS applications they use – into a central data warehouse. From there, they can run analytics, build informative dashboards, and process the data to feed into other systems with the goal of making data-driven business decisions.

Your data replication API is critical to accomplishing this task, but you may not be aware of some best practices for writing your API to ensure your users can connect to it easily and reliably.

Fivetran wrote this eBook based on our extensive experience working with SaaS application vendors' APIs. In this eBook, you will learn best practices for data replication API design that will improve your users' data replication experience.

Terminology

Before we dive into the details, it's useful to define some common terms used throughout this document so we're all on the same page:

When we say	It can also mean
Collections	Tables/Objects
Records	Rows/Documents
Attributes	Columns/Fields

Another term associated with collections is *endpoint*. An endpoint is the access point that your API provides to the records in a given collection. For example, records in a *customer* collection might be accessed via a *customers* endpoint:

```
GET https://your-e-commerce-store.com/api/customers
```

Since the full URL is not relevant to the discussions at hand, we will abbreviate the endpoint:

```
GET /api/customers
```

Another critical term is *primary key*. It refers to a unique identifier that defines the identity of a record. It can be a single attribute, or a set of attributes within the record that it identifies.

Now, let's get started:

Implement an incremental update strategy

One of the key concerns when replicating data from a SaaS application to a cloud data warehouse or lake is "data freshness." In other words, is the replicated data as current as the source data? Ensuring the destination data is up-to-date and "fresh" at all times requires frequent updating.

If your API doesn't provide an incremental update strategy, your users will need to copy all records on every update. This is not only inefficient and wasteful in terms of compute and network resources, but it also severely limits the update frequency.

In many cases, copying all the data in an application (e.g., the order history for a large, multinational corporation) can take several days or even weeks. An update frequency of once every few days or weeks is simply unacceptable. In fact, most use cases require an update frequency of once every few minutes, so having a strategy to handle updates incrementally is crucial.

In addition to adopting an update strategy, your strategy must:

- **Cover all API endpoints**, otherwise the API users must copy all records on every update for those endpoints not covered; and
- **Communicate deletions**, or the only way for the API users to discover which records have been deleted will be to copy all the records – again.

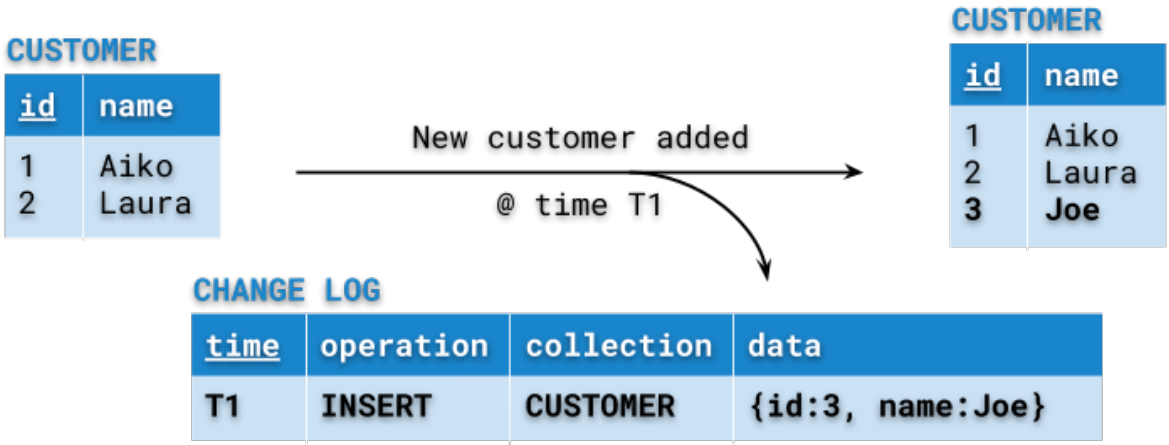
There are two best practice strategies to incrementally update your data, and we describe them in detail here.

Change Log

Using the "Change Log" strategy, changes are gathered into a time-ordered log, and any new records, updates, and deletions are included in the log. Here's what that looks like for each operation:

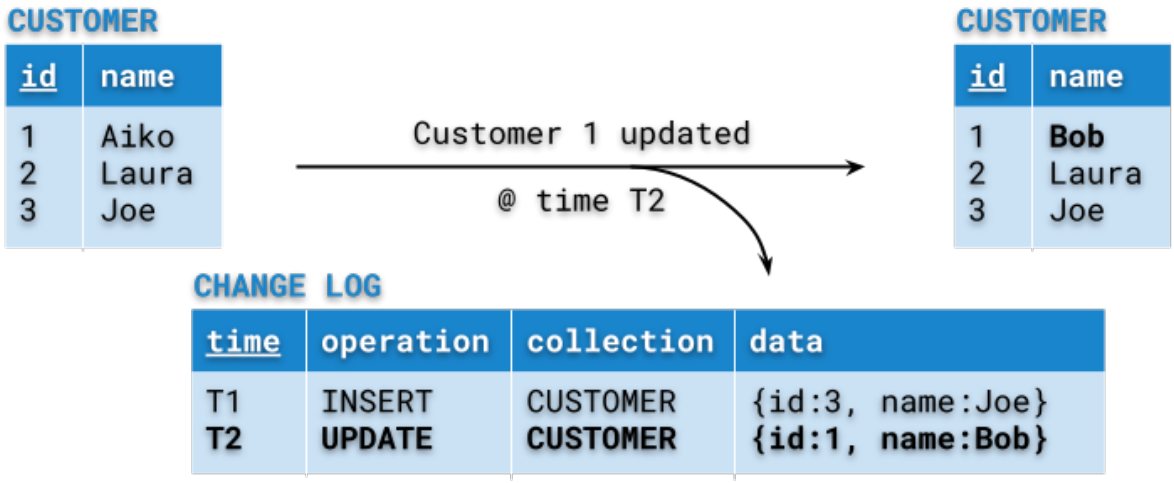
Adding new records:

When a new record is added to a collection, the operation simultaneously logs the timestamp of when the operation occurred, the type of operation, the name of the collection, and the new data, as shown here:



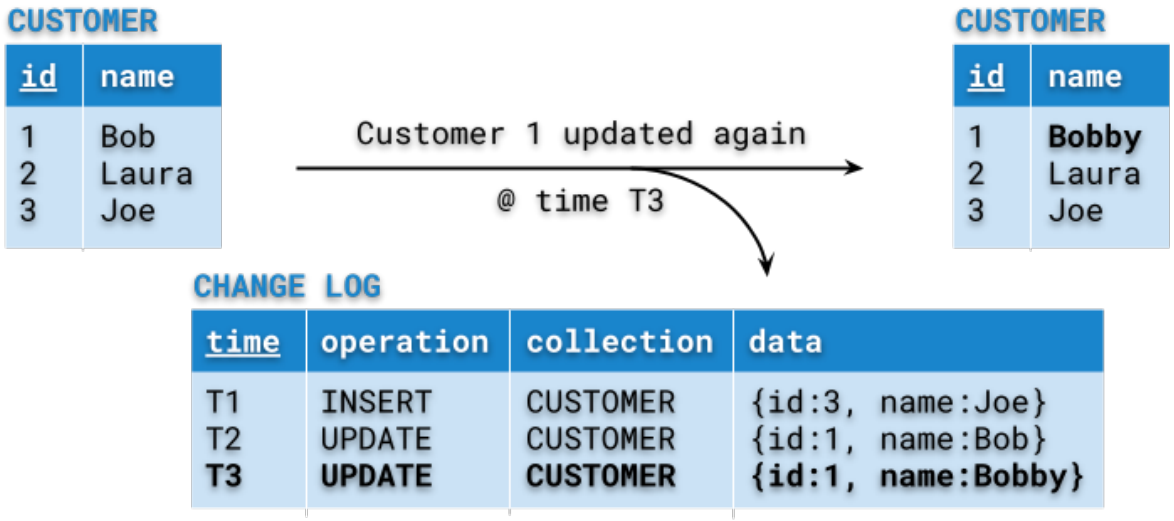
Updating records:

Similarly, when a record is updated, a new log entry is appended, as shown in this diagram:



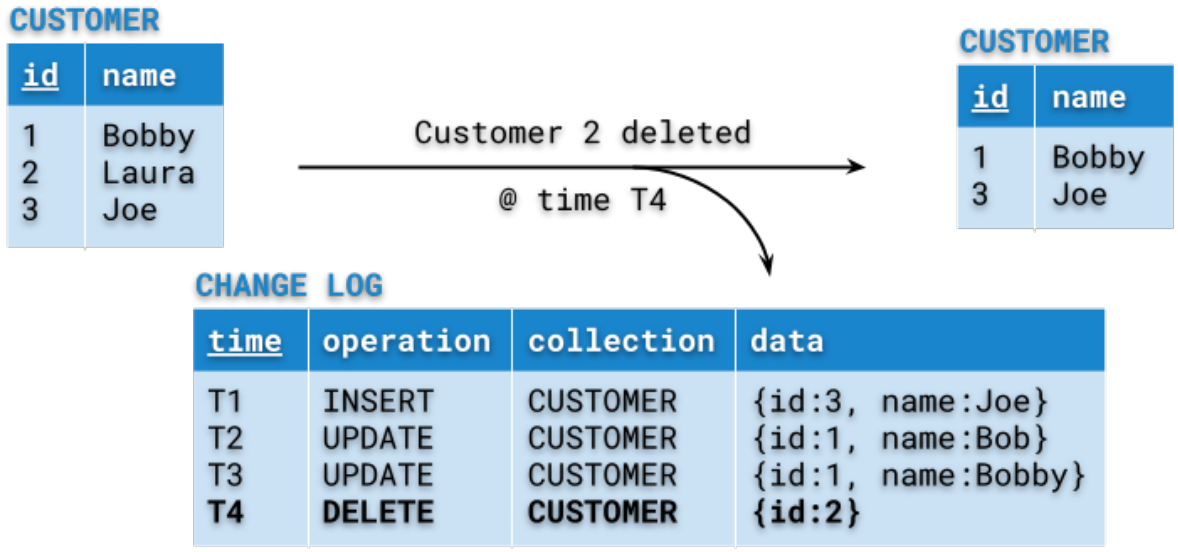
In terms of what data to log with the update operation, you only need to include the primary key and the changed attributes. This will reduce the demands on storage space for your change log as well as require less communication bandwidth for replication. On the other hand, it may be more convenient to include the entire changed record. From a data replication standpoint, either approach is acceptable.

If the same record is updated again, even if that's immediately after the previous update, the action generates another log entry that captures the entire change history, not just the final state, as illustrated here:



Deleting records:

One of the primary benefits of the change log strategy is that you don't need a separate method to communicate deletions. Instead, deletions are simply logged as part of the change log, as shown here:



It's important to note that, with delete operations, it's sufficient to log only the primary key of the deleted record.

Two other things to keep in mind about the change log strategy:

Change log format:

While presented here in a way that looks like a table, the change log can, but does not have to be implemented as a database table. It can be something else such as a sequence of files in cloud object/blob storage. What matters is that the log be efficient to read sequentially from an arbitrary starting point.

Regardless of the storage format, you can either have:

1. A **single log** gathering changes across all collections; or
2. A **separate log** per collection

The best choice is a single log because it's more convenient as well as more efficient to fetch all changes from one API endpoint.

Unique Identifiers:

It is important to make sure change log entries have a unique identifier (i.e., a primary key) of their own. With a unique identifier, your users are able to query change log entries starting from a particular point so that they can skip all the changes they have already processed and jump to the point they left off from the previous sync.

Log entries cannot be uniquely identified by the primary key of the original records, and the timestamp of a log entry isn't necessarily unique either. If you can guarantee that no two log entries can be made in the same microsecond, then the timestamp *may* suffice as a unique identifier.

Oftentimes distributed systems can easily produce multiple log entries with the same timestamp. Furthermore, clocks on different systems can be out of sync, leading to out of order timestamps in the log. Plus, data can be available some time after when the timestamp says it was recorded in the system. You need to design your distributed system to handle these edge cases properly and return log entries in the correct order regardless of how or when they are stored by your system.

If you think the timestamp is sufficient as a unique identifier, you should investigate whether it's possible for a bulk operation to create many entries with the same exact timestamp. If the timestamp alone is not unique, you'll need to construct an additional, unique identifier.

There is no need to keep log entries forever. Expiring the entries after some time is a common strategy to free up space and comply with privacy laws.

The change log strategy requires building a logging mechanism, which can be easier said than done. Often, it's simply not cost-effective to construct a logging mechanism, in which case you'll want to consider the *Modified Timestamp* strategy instead.

Modified Timestamp Strategy

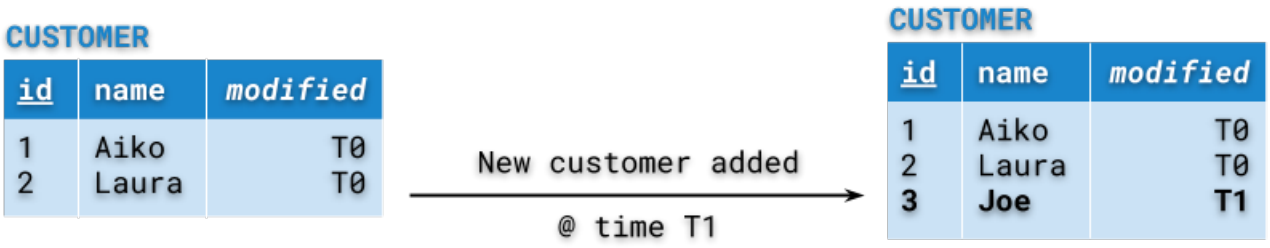
The *Modified Timestamp* strategy does not allow the entire history of changes to be replicated, but does satisfy the basic requirement of incremental updates. It relies on identifying changed rows in the original tables themselves without requiring a separate log. It does, however, require the addition of two metadata attributes to every collection: A *modified* timestamp and a *deleted* flag. You don't have to use these exact names, but be sure to:

1. Clearly document the names and behavior of these attributes; and

2. Keep the names consistent across all collections.

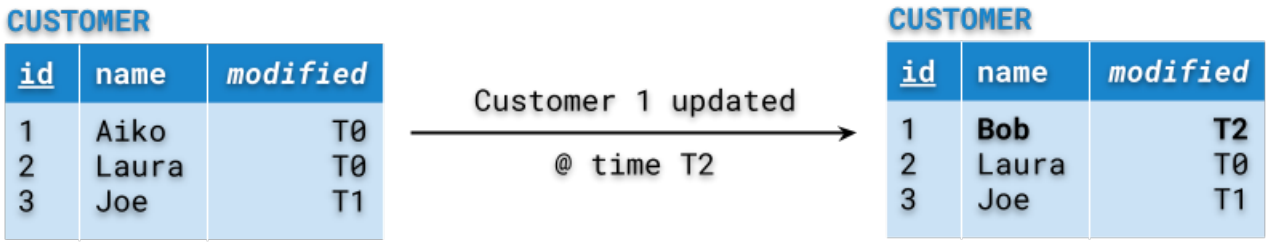
Adding records:

When a new record is added, its *modified* attribute is set to the time of the insertion operation, as shown below:



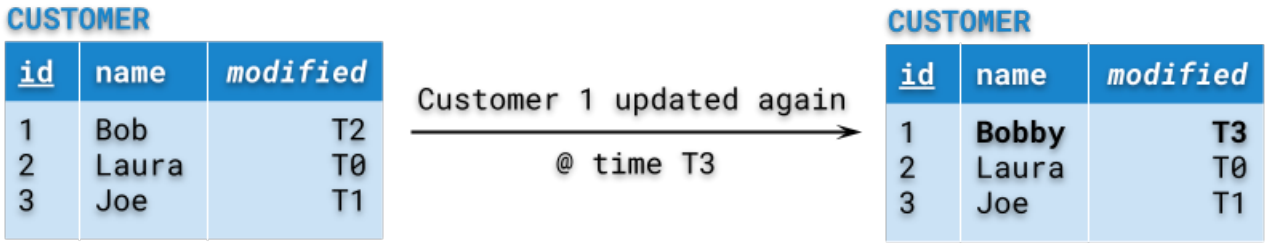
Updating records:

When a record is updated, its *modified* attribute is set to the time of the update operation, as shown here:



This makes it possible for the user, when performing an incremental update, to query only the new and changed records by filtering on *modified* timestamps greater than the time of the last sync.

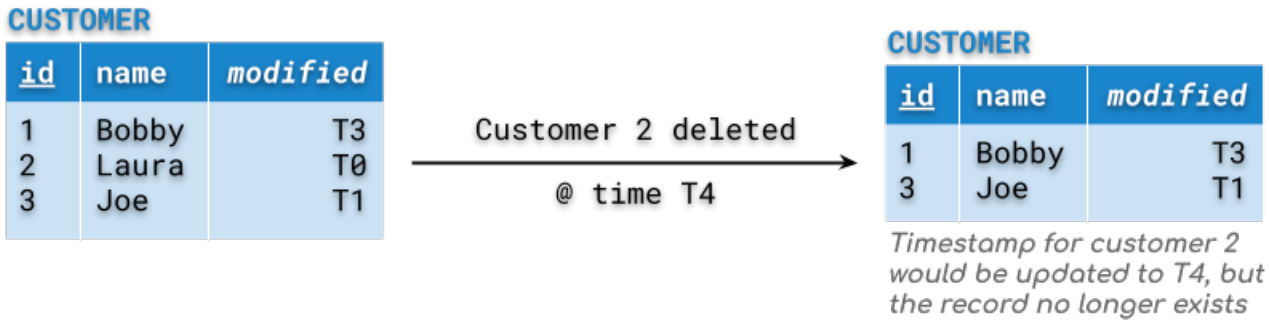
If the same record is updated again, its *modified* attribute is set to the new (latest) time of modification, leaving no trace of the previous modification, as shown here:



Therefore if a record is updated multiple times in between two syncs, then the intermediate versions of the record are lost. In other words, the *modified timestamp* strategy cannot replicate the history of changes. If you want your users to be able to replicate history, then you must implement the change log strategy described in the previous section.

Deleting records:

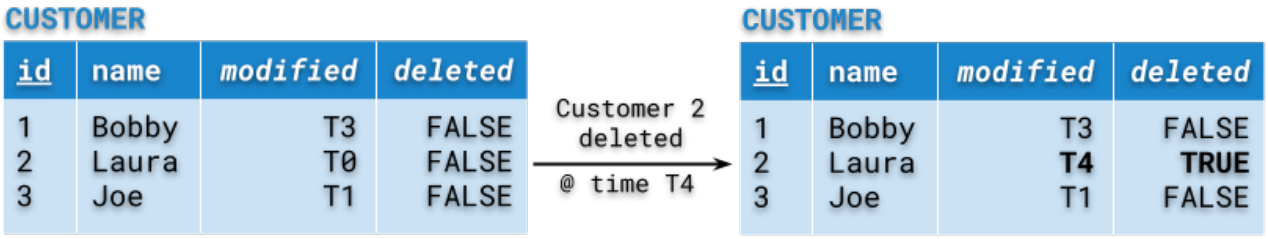
The *modified* attribute alone cannot represent deletions, because the very record containing the *modified* timestamp will be deleted, as you can see here:



This forces your users to perform a full copy of the database to discover what has and has not been deleted, defeating the entire purpose of an incremental API.

So, what's the solution? Here is where the second metadata attribute, the "*deleted*" flag, comes in. This means records are logically deleted by setting *deleted* to TRUE but are not actually deleted. Another way to look at this is that deletions are treated as updates.

In addition to setting the deleted attribute, the modified attribute is set to the time of deletion, as shown here:



To eliminate any doubt, the *deleted* column is required, not optional. Without it, the modified timestamp strategy would be incomplete and render the API non-incremental.

This strategy is only effective if the *modified* timestamp is updated every single time any change is made to any given record. To ensure this is done systematically without missing any cases, you should automate the updating of this timestamp within the database itself rather than in the application logic. This can be accomplished, for example, using triggers.

To maintain your API's performance, remember to create an appropriate index (e.g., B-tree) on the *modified* attribute, so that records modified after a given time can be efficiently fetched using a range scan rather than a full table scan.

One drawback of the *modified timestamp* approach is that application logic must compensate for the fact that the "deleted" records are not actually deleted. For example, when fetching a list of all customers, records in which *deleted* = TRUE must be filtered out. If changing the application logic is impractical, you may prefer instead to implement the *change log* strategy described in the previous section. The *change log* strategy is superior anyway since it captures the entire change history rather than just the latest state.

Now that we've laid out the most important part of replication API development —the update strategy— here are some other general guidelines:

Provide a fast, straightforward way to fetch all data

While an incremental update strategy is critically important, incremental updates are only relevant after an initial copy of all the data has been replicated. When a user first connects to your API, they'll want to quickly and efficiently make a full copy of all records in every collection. This is also known as an initial, historical sync.

To enable this, your API must:

1. Provide a complete set of endpoints

You need to allow access to ALL data in the user's account, even the data that seems to be esoteric or rarely used. By iterating through all the documented endpoints and fetching all records from each endpoint, your user should end up with a complete copy of the data in their account.

2. Iterate quickly through all data

Your users must be able to fetch all records in all collections in a reasonable amount of time. The number of records in a given user's account depends largely on the size of their business. This can vary from thousands of records for a small company to billions of records for a large company. For small accounts, iterating through and fetching the full data set should take no more than a few minutes. For large accounts with billions of records, it should take at most hours or days rather than weeks or months. To make the underlying implementation efficient enough, you should consider the following:

A. API endpoint design: Make sure records can be fetched many at a time from any given collection. Don't make your users fetch one record at a time. We have seen this mistake often with sub-records, for example when an order record contains ID-s of line items, but the line items can only be fetched one at a time.

B. Throttling: Although every good API includes throttling as a means of protection, the throttling must not be overly restrictive or it will make it too slow to iterate through all records. Document the throttling limits and inform your user programmatically through a 429 response when throttling kicks in.

C. Indexes/Paging: See the section below on breaking up large responses

D. Avoid joined data: See the section below on this

Be sure to stress-test your API to make sure a large account can be replicated expediently!

3. Eliminate duplicate data

A given set of records should only be available through one endpoint, not through multiple different ones. For example if there is a set of addresses associated with each customer, these addresses should be available EITHER as a set of nested records within customer records from the *customers* endpoint, OR separately through an *addresses* endpoint. The addresses should not be available through both endpoints because that causes confusion in two ways: a) It's not clear whether the addresses from the two different endpoints are actually different or simply duplicated, and b) If duplicated, then which endpoint to use since one is probably much less efficient than the other. The more efficient option depends on how the data is structured in the underlying storage, so make sure to offer only the more efficient option.

In some cases, there may be a reason to represent two seemingly identical sets of data as separate. For example in an e-commerce store, the address associated with an order can, but does not have to be, the same as the address of the customer who placed the order. The customer might enter a one-time address for a given order or change the address on file, and this should not change the address associated with past orders. In this case order addresses and customer addresses should be separately available from the API because they are, in fact, independent. Be sure to clearly document the reasons why.

Break up large API responses into pages

It is important to break up large API responses into pages because large responses are a leading reason for failed data transfers. The larger an API response, the longer it takes to complete a single request. The longer it takes to complete a request, the higher the chances that the request will be interrupted by a timeout in some system layer – either on the client side or the server side, or by a random networking failure. If this happens, the entire request must be retried. Both the probability of failure and the time cost of failure increase with larger response size. In some cases, the request is too large to ever successfully complete, resulting in endless retries.

Breaking up the set of returned records into smaller, more digestible "pages" helps increase the success rate of each individual request and helps ensure that, if a request does fail, the user doesn't need to start from the beginning and fetch the entire set of records again.

While breaking returned records into smaller pages matters both for initial, historical syncs as well as for incremental updates, it's particularly relevant for historical syncs because that is typically when large volumes of data are replicated.

For example, if the *customer* collection contains seven million records, and the user wants to fetch them all:

```
GET /api/customers
```

The API should not attempt to return all seven million in a single response. Instead, it should set a maximum number of records that a single response can contain, e.g., 1,000 records per page. The user would then fetch the first page of 1,000 records in the first query, the next 1,000 records in the second query, and so on. They will have to make seven thousand separate queries to fetch all seven million records.

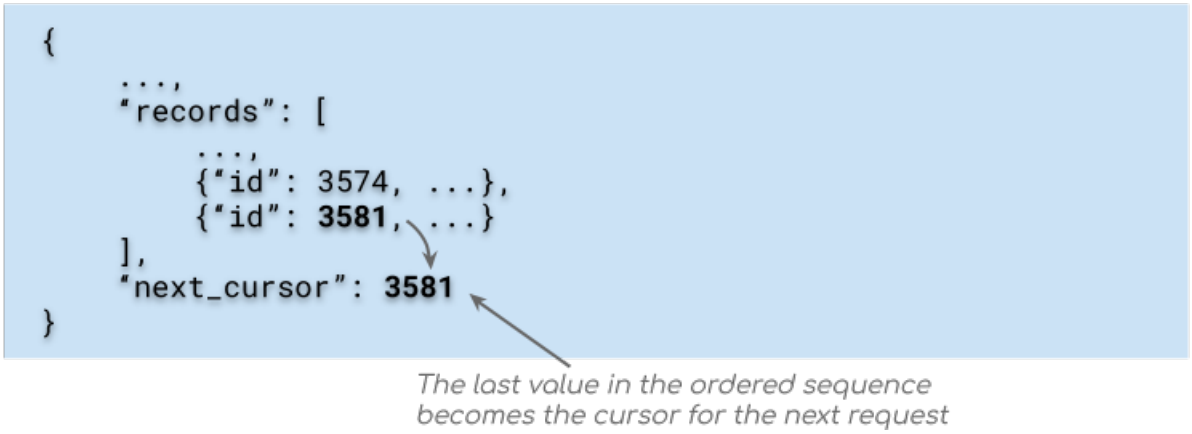
There are three common approaches to paging:

1. **Paging Cursor** - the best
2. **Paging Key** - second best
3. **Page Number** - not worth it, don't do it!

In the sections below, we examine each of these approaches in detail, including how to do it right, and how not to do it.

The best approach: Paging Cursor

The best approach uses a cursor that points to a unique place in the sequence of returned records based on a unique, ordered attribute. For example, the primary key ID often makes a good paging cursor. If the results are in primary key order, then the ID of the last record in a page of results can be used as the cursor for the next request, like shown here:

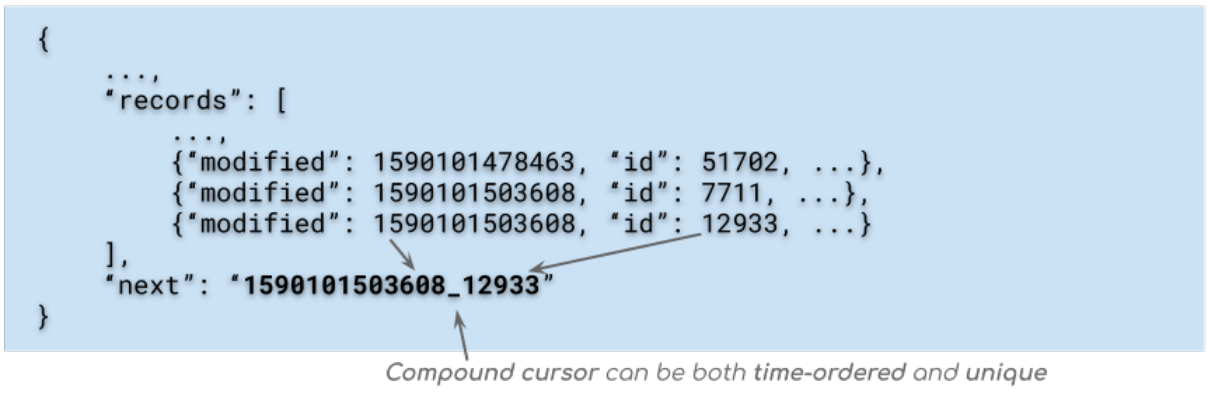


In a SQL backend, this would map to a query like this:

```
SELECT * FROM CUSTOMER WHERE id > 3581 ORDER BY id LIMIT 1000;
```

Of course, the results must be ordered by the cursor attribute, which is, in this case, the primary key. In some databases, such as MySQL, data is natively ordered by primary key. In other databases, you may need to create a scannable index to make sure paging will operate efficiently.

You might want different cursor implementations when you want to order the results by something other than the primary key. For example, if your API uses the last-modified update strategy, you might want to return the results in order of the *modified* timestamp. Because several records can have the same timestamp, you need an additional attribute to uniquely identify the records. In this case, you might use a timestamp-primary-key combination, encoded into a single string:



Explicitly representing the next cursor in the response is important because it allows you to have different implementations of the cursor without needing to change the API contract, which gives you the freedom to change the implementation in the future or to have different implementations for different API endpoints. Your users get the benefit of handling paging on their side using a single, shared piece of code across all endpoints that never needs to change.

Make sure your users can iterate through the results quickly and efficiently. This can be achieved by building a scannable index on the combination of both attributes. A detailed example of a MongoDB-based implementation can be found here.

If your particular application calls for something other than the examples above, the fundamental requirements to keep in mind are:

1. A cursor value should uniquely identify a single record
2. Results should be ordered by the cursor attribute
3. Iterating through the results should be fast and efficient

Second-best approach: Paging Key

This approach works similarly to the paging cursor approach, but instead of calling the cursor a "cursor", it uses the name of the underlying attribute as part of the query parameter, e.g. if paging by primary key attribute called "id", then the name of the key attribute is directly used in the query parameter:

```
GET /api/customers?after_id=3581
```

Request records, starting immediately after this ID

To get the value of the cursor for the next page, the user can simply look up the value of *id* in the last record of the previous page.

```
{
    ...,
    "records": [
        ...,
        {"id": 3584, ...},
        {"id": 3587, ...}
    ]
}
```

The last ID becomes the cursor for the next request

The wrong approach: Page Number

A common but problematic approach is to use *page* number as a query parameter:

```
GET /api/customers?page=3
```

DO NOT DO IT THIS WAY

Page number selects which page of results to fetch, but some records might be skipped with this method!

This is an unsound approach because page numbers would have to be implemented in the backend via offsets. For example, let's assume a page size of 1,000 records. If the backend is a SQL database, the query that fetches page 3 might look something like this:

```
SELECT * FROM CUSTOMER LIMIT 1000 OFFSET 2000;
```

Skipping the first 2000 records gives us the 3rd page, but the results are not necessarily stable

However, in an actively used application, new records could be added or some records removed after querying page 2, but before querying page 3.

Further, if the removed records are among the first 2,000 records, then all subsequent records are shifted down, which means the first few records of what would have been page 3 now become the last few records in page 2, and therefore will be skipped when querying page 3.

The page-numbering approach may work for append-only tables, but do you really want to bet that your tables will be forever append-only?

Even if you win that bet, using OFFSET has the downside of causing the underlying database query to run slower as the page numbers increase because the database has to scan through the table to get to the offset position.

In the discussion above, we showed how to implement paging for any given endpoint. Since an API contains many endpoints, remember to implement paging consistently for all endpoints!

Define all keys

Much of what has been discussed in the previous sections depends on the ability to uniquely identify records. Not only do **Primary key IDs** make it possible to identify records to update or delete as part of an incremental update strategy, but they also make it possible to implement paging correctly without skipping any records. And, in the event that the same record is ever returned twice, primary keys enable deduplication.

In order to realize the benefits listed above, a primary key value must:

- *Uniquely identify a record* -- No other record can have the same primary key value, now or in the future)
- *Never change* -- While other attribute values in the record can be updated, the primary key value must always remain the same to ensure that the correct record is updated in the replicated data destination. For example, a customer's email address wouldn't be a good primary key because it may need to change, which would lead to a second record being created in the destination for the same customer.

A primary key ID must exist for every record, and the primary key attribute needs to be clearly identified in your API documentation. Ideally, the primary key is something simple such as *id*, as in the examples above, and it is consistent across all collections. If it isn't consistent, then it must be separately documented for each API endpoint. Don't leave your users to guess the name of the primary key because it's not always obvious. For example, suppose a user receives the following record from an *orders* endpoint:

```
{
    "order_id": 1,
    "order_number": 38,
    "submitted_at": "2020-05-22T06:16:34Z",
    ...
}
```

If only the field *order_id* existed in the database, maybe it would be easy for the user to guess accurately. But if you add in the *order_number* field, suddenly it's not so easy to figure out. Therefore, it's important to always document the primary key, whether it's obvious or not.

To make matters even more interesting, consider the following record returned from an *order_items* endpoint:

```
{
    "item_id": 119,
    "item_number": 5,
    "order_id": 1,
    "description": "T-shirt",
    "product_id": 78
    ...
}
```

Does *item_number* uniquely identify an order item among all orders or does it require a compound primary key of *order_id* and *item_id* to identify an item? Or maybe *order_id* and *item_number*? We can't stress this enough: make sure to define the primary key in your API documentation.

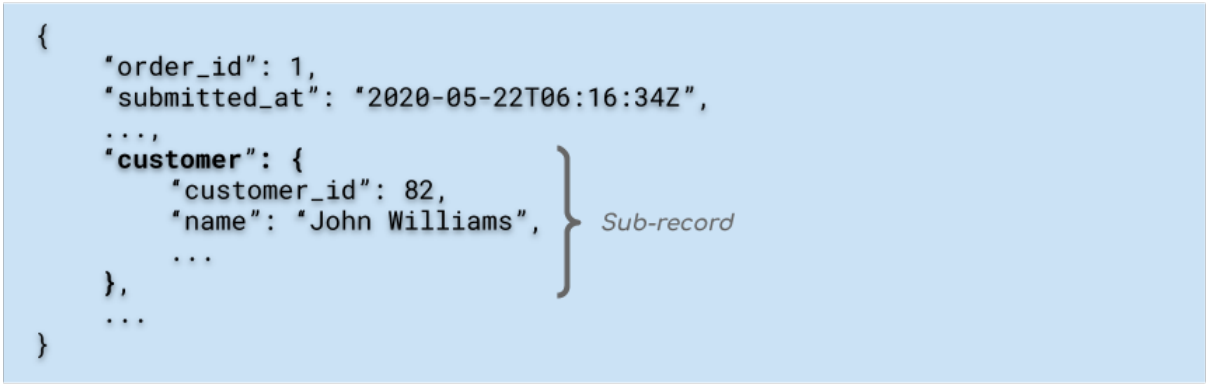
Finally, the *order_id* and *product_id* attributes in the above example are presumably foreign key references to other collections. Each of them likely refers to records in the *order* and *product* collections, respectively.

It bears repeating: Don't leave your users guessing!

Define all **foreign key references** in the documentation because your users are syncing this data into their data warehouses and they want to use the data for analytics. In order to perform the correct joins on the data, users need the foreign key information.

Avoid joined data

Speaking of joins, sometimes API responses include sub-records and sub-collections within records. For example, consider a record from the **orders** endpoint that looks like this:



The sub-record within the *order* record contains information about the customer who placed the order.

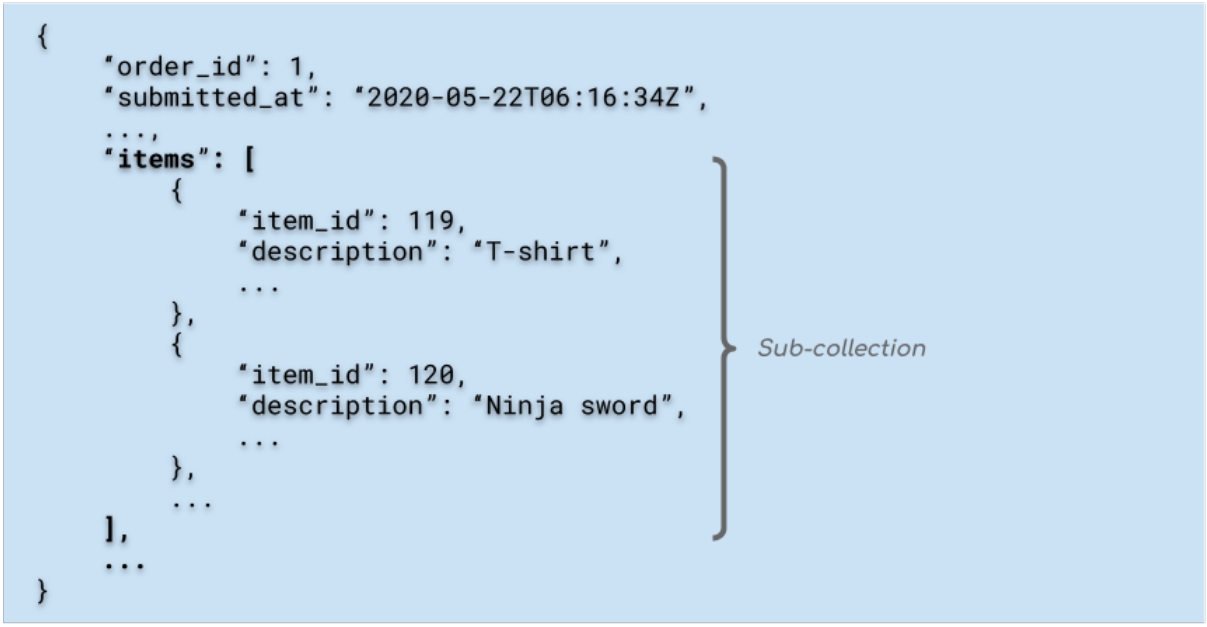
Customer details are not usually stored with order details. Assuming there is a separate *customer* collection in the underlying database, customer information should be returned from a separate *customers* endpoint, not joined onto the records returned from the *orders* endpoint. Joining information from two separate collections makes the API unnecessarily slow for three reasons:

1. Join operations are inherently very slow compared to a simple, sequential scan of a single collection;

2. Processing and network bandwidth is often wasted fetching information about the same customer multiple times (assuming there are multiple orders per customer); and

3. Even more time and bandwidth is wasted fetching all the customer records again from the separate customers endpoint otherwise, customers that did not place any orders will be missed.

On the other hand, sometimes sub – records or sub – collections are an integral part of a parent record. Consider a case in which the *items* that make up an order are included as a part of the *order* record:



In this case, it depends on how the data is stored in the underlying database. For example, MongoDB allows nested records and collections, so the items would likely be stored as nested inside the order to which they belong. On the other hand, relational databases, such as Postgres or MySQL, would store order items in a separate table.

The bottom line is that data returned by the API endpoints should reflect how the data is stored. Return nested data if it is stored as such, but don't include any nested data if that would require joining separate collections.

Follow a consistent set of rules

It's surprising how many APIs follow some, but not all, of the above principles – and even then, inconsistently. For example, some APIs implement last-modified timestamps for some collections and not for others.

Consistency is key to successful implementation of all the above principles. Everything from the update strategy, paging, and naming must be consistent across all parts of the API. Even elements as seemingly inconsequential as names of query parameters and names of metadata fields returned in the query responses must be consistent.

Why is that important?

Imagine if one endpoint returned a paging cursor named *next_cursor*:

```
{
    "records": [
        ...,
    ],
    ...,
    "next_cursor": 89231
}
```

While another endpoint returned its paging cursor under a different name *next*:

```
{
    "records": [
        ...,
    ],
    ...,
    "next": 3412303
}
```

Seems like an insignificant difference, but the API user must now have a branch in their code using one name for one set of endpoints and a different name for a different set of endpoints. Plus, the user must now maintain a list of endpoints that behave one way and another list for those that behave the other way, just for this one tiny inconsistency. Imagine if your API included several inconsistencies – you'd end up with a very convoluted implementation on the user's side.

Keeping everything consistent makes for cleaner code on both sides – and cleaner code leads to a more reliable implementation.

Clearly document the API

Any lack of clarity, omissions or mistakes in API documentation lead to bugs and errors in implementing data replication, resulting in a frustrating user experience. Therefore, all aspects of the API, as discussed in this document, must be clearly documented, including:

- Update strategy
  - Name of the last-modified timestamp attribute
  - Name of the is-deleted flag attribute
- Paging strategy
- A list of all collections and their respective API endpoints
- For each endpoint:
  - All attributes returned by API responses, including the data type, description of meaning, and example for each
  - All available query parameters (both mandatory and optional)
  - Name of primary key attribute
  - Names of foreign keys and what they refer to

Conclusion

A well-designed replication API is an integral part of your product and deserves the same level of care as any other part. Your users will want your application to be part of a data ecosystem rather than remain isolated. They will benefit from their data being integrated with the rich cloud-based data analysis tools that are now available. Such integration is performed by a data pipeline. The pipeline can only be reliable and efficient if it is connected to a source with a well-designed replication API.