Functional Data Pipelines and OLAP

Performant Analytics Using Hybrid Data Stores

Dexter Taylor, Principal, Binary Machines LLC
dtaylor@binarymachines.io

The construction of high-performance data pipelines for analytics involves the careful balancing of numerous concerns. We have found that it is possible to build very high performance pipelines using hybrid SQL/NoSQL systems and utilizing some of the principles of functional programming, without resorting to resource- and maintenance-intensive solutions such as the Hadoop ecosystem. We call pipelines constructed according to these principles HFP's, or Hybrid Functional Pipelines.

The relational database management system, or RDBMS, is the concrete realization of design principles first set forth by E.F. Codd and Chris Date in the late 60's/early 70's. The seminal work on these principles is Codd's 1970 book "A Relational Model of Data for Large Shared Data Banks". Relational theory, which is a relative of set theory, proposed a ruleset for structuring data based on set membership and designed to minimize the cost of updating records. Databases build around these rulesets quickly became the de facto standard for business data management; to this day, most of the world's business data is stored in relational databases.

These databases have a few important characteristics in common:
- They are queried using SQL
- They require pre-defined schemas (also specified using SQL) in order to store data
- They use ACID (Atomic, Consistent, Isolated, Durable) transaction semantics for write operations

The tradeoff here is that we accept higher write latencies and a priori constraints on the format of records to be stored, in exchange for greater data reliability and self-consistency.

More recently, a generation of databases has emerged which does not use the relational ruleset; these are colloquially known as NoSQL databases (although some of them do allow queries using a dialect of SQL). Much has been made of the "higher performance" offered by NoSQL databases, the implication being that defining data schemas and querying stored data using SQL is itself inimical to performance.

The truth is significantly more complex; relational databases are capable of very good

performance if designed, provisioned, and tuned properly, and NoSQL databases can give quite bad performance if correct practices are not observed. A good deal of the performance penalty incurred by relational databases is a write penalty; the logic that governs ACID transaction semantics is nontrivial and at least in part dependent on disk I/O, which can blunt the advantage of faster processors and more memory. A database which holds all its records in memory, for example, would be a good deal faster (all things being equal) than one which saved records to disk; the tradeoff being that catastrophic power failure would result in 100% data loss. Likewise, a database which does not enforce ACID transaction semantics can be significantly faster than one which does not, with the caveat that multi-client, update-intensive operations run a much higher risk of introducing errors and inconsistencies into the data.

When we consider data analytics, we are usually considering the question of how to efficiently deal with business data which is not in the day-to-day, front-office transaction processing stream. This gives us more flexibility with respect to how we structure and store data, because:

- Analytical (as opposed to online-transactional) data operations are read-intensive, not write-intensive
- We can reload data into an analytical store at will, so we can use more ephemeral datastores; in such cases, some amount of reload time is the only penalty for catastrophic failure of an analytics computing system

If we further stipulate the functional-programming precept of *immutability* as a core feature of our pipeline design, we reap the additional benefit of being able to use relational databases in a read-only way, which gives us:

- The ability to use an already well-understood query language (SQL) and a wealth of preexisting reporting tools
- A way to bypass resource-intensive transaction semantics most of the time; our database becomes a WORM (Write Once, Read Many) datastore

Design Metaphors and Patterns

In an HFP (Hybrid Functional Pipeline), we do many of the tasks which are common to other pipelines; we still must perform some sort of ETL (Extract, Transform, and Load) on data, and we must provide a mechanism for querying processed records.

The key concepts that characterize HFP's are:

- Separation of concerns between so-called raw records and cooked records
- Immutability (updates are actually inserts, so no data is ever overwritten)
- Journaling (the pipeline keeps a granular record of its own operations)
- Data operations are linear in the core of the pipeline and two-dimensional only at the edges
- Out-of-order operation

The Raw and The Cooked

A **raw** record, for our purposes, is one which is structurally correct but may still contain data errors; an example would be a sales record whose date format is correct, but in which the date itself is wrong. A **cooked** record is one which has been corrected and from which we can generate a record in the output datastore. (Note that **input** records, which come directly from our source data, may not always make it into the pipeline; a record which is structurally incorrect should not be admitted as raw, but should instead be stored in a holding area and logged as could-not-process. This allows core pipeline logic to be simpler, as it can proceed from a baseline set of assumptions about record structure.)

An HFP requires strict separation of concerns between the segments of the pipeline which handle raw records, and those which handle cooked records. If we map this onto the traditional ETL data processing paradigm, we would say that E, T, and L are each discrete modules mostly decoupled from each other. Extract does not care about transforming records, but only passes structurally-correct data to the Transform module. Transform does not care where records came from – disk, memory, network – but only concerns itself with correcting errors or adding specified enhancements to business records.

This separation of concerns also means that we can write multiple Extract modules: one for each external data source. Extract communicates with the other modules in the pipeline solely by writing set of name-value pairs representing the source record (plus some metadata) to the pipeline datastore.

Immutable Data

Within the pipeline core, updates are actually inserts; the logic in Transform does not change the records passed by Extract. Rather, it copies them into its own portion of the datastore and applies updates to the copied records. In this way we can ensure that no

two pipeline clients will ever contend to update a record; we never need lock records, and there is a built-in audit trail of what changed and when.

Once records have been "updated" by Transform logic, they (or more accurately speaking, their keys) are handed to the Load module, which is solely responsible for writing the records into a datastore optimized for the specific analytical queries desired by stakeholders.

Journaling

Each of the three basic operation types (E, T, L) performed by a hybrid functional pipeline is journaled, using a suitably performant NoSQL database (we prefer Couchbase for its ease-of-use, scale-out ability, and SQL query semantics). Tasks should be journaled when they are started and when they are completed, and each journal entry should contain, at minimum, the operation type under way, the UNIX process ID of the initiating program, a timestamp, and a task completion status.

In this way, we accumulate useful metrics simply as a side effect of operating the pipeline; for example, we can generate a simple heat map or performance profile by issuing a SELECT * for all the journal records where the delta between the start time and completion time exceeds some threshold.

Additionally, if a long-running process in our pipeline dies, we can know approximately where it failed and take steps to either purge and re-import missing data, or backfill the datastore before omissions cause problems in the downstream analytics.

Linear vs. Two-Dimensional Data Operations

A linear, or one-dimensional, data operation is one that returns a list of records based on a simple spatial or extrinsic predicate: for example, "give me the next 10K records from the input queue" or "show all records which were ingested in the last 30 minutes". A two-dimensional data operation, by contrast, is more like a *query* as commonly understood. It is typically issued against a database and returns an answer set based on intrinsic and possibly complex predicates; for example, "give me all the sales records where the amount paid is greater than X".

Data operations at the edge of the pipeline can be two-dimensional, but operations in the core of the pipeline should, to the extent possible, always be linear.

The essence of this design feature is that *no logic in the core of the pipeline should ever execute a query*. The reason, simply put, is performance. Linear operations can return the correct answer set without introspecting into the data and without engaging a potentially expensive query-language runtime. By contrast, a query needs to be marshaled across a language binding or other run-time interface, parsed and possibly optimized by the query planner, and executed by the database engine – after which the answer set needs to be transmitted back into the caller's process space.
Needless to say, even in a well-tuned database, this involves latencies which can greatly exceed that of a relatively simple data copy. When we are running analytics, those latencies are simply the cost of doing business. But note that the purpose of a data pipeline is not to perform analytics; it is to capture and transform records and write them into a datastore against which *our end users* perform analytics.

In cases where queries are unavoidable for whatever reason, those queries should be performed in advance (say, when the pipeline is spun up) and the results cached in memory, where they can be accessed using a simple lookup.

Out-Of-Order Operation
We have seen a variety of ETL "workflow" tools (such as Luigi and Airflow) designed to guarantee the in-order execution of a series of steps in a data pipeline. The execution order may be a simple linear succession, or a more complicated flow represented by, say, a directed acyclic graph. Where such tools are in use, there is often a sensitive dependence on not just the order of execution, but the presence of secondary execution artifacts (for example, logs or intermediate data-output files) which must be present after the completion of one step in order for subsequent steps to run.

We find that while such tools may sometimes be genuinely necessary, their use is often a sign of flaws in the underlying pipeline design. One of the hallmarks of a good design is the ability of each of the steps in the pipeline to execute out-of-order; that is, with no consideration as to which step (if any) ran prior, or which will run subsequently.

HF pipelines accomplish this design objective using a combination of two technologies: NoSQL key-value databases and high-capacity in-memory datastores such as redis.

For example: say the Extract module has pulled a source record and performed some initial checks on it, and is now ready for it to be transformed. In our theoretical HF pipeline, what would happen is that Extract would write the record to its core datastore (say, Couchbase) and get back a unique key. It will then write that unique key to a redis

queue, which implicitly informs the next stage of the pipeline that there is a new record waiting to be transformed.

When the Transform module executes, it pulls a linear dataset (a list of keys) from the in-memory queue and deals with them in order. But note that the Transform module need not be explicitly notified, either by IPC or by a secondary artifact such as a signal file, that there are records ready to process. Upon execution it should always do the same thing: ask the in-memory queue for a set of keys. A nonempty set is a signal that there is work to do; an empty set is simply a short-circuit condition, prompting either a normal shutdown (in the case of a *cron* job) or a sleep for the specified polling interval.

Pipelines which are brittle with respect to execution order are most often found in enterprises employing batch-processing methods. Continuous real-time, or "streaming", data pipelines -- which may use producer/consumer queues to pass records between processes -- are a different matter; but there we see quite clearly the importance of processes which are completely insensitive to execution order. Data processors in streaming pipelines are much closer to being simple event handlers, and it is axiomatic that a properly-designed event handler is completely decoupled from other handlers, even those in the same process space.

Case Study: HFP In Practice

A major Swedish online retailer needed sales data analytics. Their starting condition was that transactional sales records were offloaded daily to a data lake on Amazon Redshift; some of those records contained price information that was incorrectly calculated due to differences in local fees and taxes inside and outside the EU. Their analysts were familiar with desktop cloud-based tools such as Mixpanel and Sisense. What they wanted was a pipeline connecting their data lake with an analytics database which could be queried directly by the tools their analysts were already using.

We provisioned a group of machines in the virtual datacenter (an AWS Virtual Private Cloud) consisting of:

- A Couchbase database cluster
- A Redis instance
- A "coordinator" instance hosting all the Python scripts representing the pipeline logic

In addition, we provisioned a target database (also on Redshift) which featured a handful of OLAP star schemas. The most important of these was the order_lineitems schema, containing:

- A single fact table representing a denormalized set of orders and their constituent lineitems
- Dimension tables representing the originating country for each order, the hour of the day in which the order was placed, the order date, the SKU of the lineitem, and the vendor ID for the SKU

We generated extract, transform, and load scripts as discrete modules to operate the pipeline. Each script was capable of being run independently and out-of-order, and the transform and load scripts were also capable of being run in parallel (multiple transform and/or load processes executing simultaneously). All processes journaled their activity to a Couchbase bucket designated for logging, and used a Couchbase-hosted Memcached bucket for accessing frequently-used data.

The extract script performed data extraction from Redshift to temporary local storage, then read records from local storage into Couchbase; each CSV file containing N valid records resulted in N raw data documents in a Couchbase bucket and N corresponding document keys in a named Redis queue.

The transform script read keys from the Redis queue, serially retrieved the corresponding Couchbase records, transformed them in-memory, and wrote the results into a new bucket, also in Couchbase. For every transformed record written, the transform script would write the corresponding key to a Redis queue for cooked records.

The load script retrieved the cooked records from Couchbase (again, pulling their keys from Redis and performing a simple key-value lookup), accumulated the output records in a CSV file, and issued bulk-insert instructions to the OLAP datastore in Redshift.

From there, analysts on the ground were issued credentials to connect directly to the Redshift datastore and were able to use the tools of their choice. Because of the nature of

OLAP schemas, these queries were highly performant, yet easy to reason about and debug.

The delivered pipeline easily met the specification; on an under-provisioned cluster, without tuning, we clocked its performance at ~10 million records per 4-hour overnight run, which exceeded the client's daily sales volume. Additionally, the client was now ready to move from batch to streaming data ingest if the need arose. Because the transform and load pipelines had no knowledge of where source data came from or how it arrived, we could upgrade the command-line driven extract script to ingest records in real time simply by wrapping it in an HTTP-based microservice – with no change in how it communicated with downstream pipeline processes.

Hybrid Functional Pipelines utilize the most desirable aspects of modern SQL and NoSQL datastores to good effect. They allow data consumers to remain in the toolsets that give them the highest productivity, while allowing systems designers to trade storage and memory for raw performance. Their fundamental design principles are those of all good software: separation of concerns, modularity, immutability, and conscientious logging and recordkeeping. Also, the use of cluster-based key-value datastores in the pipeline core allow HFP's to scale-out to a remarkable degree without forcing teams to invest in complex, high-maintenance solutions.

Dexter Taylor
Principal, Binary Machines