NVIDIA

ACCELERATING CUSTOMER CHURN PREDICTION

An End-to-End Blueprint for Building Real-World Machine Learning Systems

Accelerating Customer Churn Prediction

An End-to-End Blueprint for Building
Real-World Machine Learning Systems

William Benton

Contents

Chapter 1

Introduction

If you want to learn how to build a predictive model to solve a particular kind of business problem, you'll likely have no trouble finding a tutorial showing you how to extract features and train a model. This is a pleasant side effect of the popularity and importance of data science and machine learning across industries. However, solving business problems with machine learning isn't just about training models or even about finding the best features; if you want a starting point for solving an entire problem from end to end, or if you want a realistic production workflow to test your system architecture or hardware performance, these tutorials leave many of the hard parts as an exercise for the reader.

In this book, we're going to build up a complete solution for predicting customer churn — that is, answering the question "given what we know about this customer, is she likely to not renew her contract." We'll pay special attention to parts of the process that are underserved by most data science tutorials, such as analytic processing and federation of structured data, integrating enterprise data engineering pipelines with machine learning, and production model serving. We'll also show you how to accelerate each stage of this workflow with GPUs.

We're going to start by level-setting: introducing a typical end-to-end machine learning workflow, describing some of the challenges of production machine learning systems, and explaining some of the specific concerns in understanding customer churn. We'll then introduce the overall architecture for our churn prediction solution.

1.1 Data science workflows and the customer churn problem

If we were talking about data science ten years ago, we'd have been referring to a broad discipline that combined domain expertise with elements of analytics, applied statistics, machine learning, computer science, and software engineering. A data scientist might have had to wear many hats: identifying business objectives; cleaning data; processing big data at scale; identifying features; encoding features; selecting and training models; building applications, reports, or dashboards that incorporate those models; and even managing infrastructure! Today, a typical data scientist is more specialized and only focuses on parts of the classic data science workflow: characterizing data, finding patterns, and training models. The other parts of the classic data science workflow are still important, but the practitioners who are responsible for them may have a range of titles other than "data scientist," such as data engineer, application developer, machine learning engineer, or MLOps engineer. We'll introduce such a workflow now (borrowing concrete terminology from this paper) in the context of the churn prediction problem.

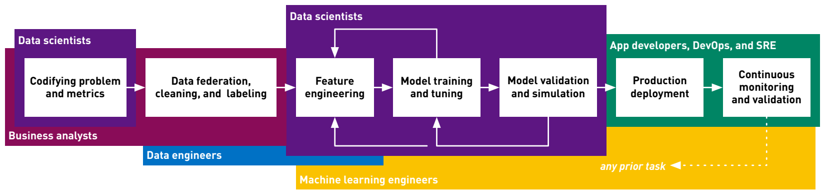

Data scientists
Codifying problem and metrics
Business analysts
Data federation, cleaning, and labeling
Data engineers
Data scientists
Feature engineering
Model training and tuning
Model validation and simulation
App developers, DevOps, and SRE
Production deployment
Continuous monitoring and validation
any prior task
Machine learning engineers

Figure 1.1: A typical data science workflow, showing the human processes from discovery through production, and the team members and roles involved in each stage.

Codifying our problem and defining metrics

The first step is *formalizing the problem we're trying to solve and defining metrics of success*. On a long enough timeline, every customer will fail to renew their subscription, but producing a model that asserts "yes, eventually" for any customer isn't useful. Similarly, we might be able to predict quite accurately that a customer who has begun the process to cancel her account is quite likely to churn in the near future, but such a prediction is also trivial. Ideally, we'd want a model that identifies signals in a customer's profile that indicate that the customer may churn in the near but not immediate future and is thus a good candidate for targeted retention efforts. (We may also want to rank customers for more costly targeted retention efforts, e.g., by expected future lifetime account value.) The exact details of how we formally define churn will be specific to our business and the kinds of retention efforts we may have to offer; given such a definition, though, we can label historical customer records and build a target for our model.

Data federation, cleaning, analytics, and labeling

Our focus next shifts to *identifying, transforming, and federating* potentially-relevant information about our customers. Ideally, we will be able to draw upon structured data, such as transactional databases, and unstructured data, such as call center transcripts. Federating this data in a single logical location, like a data lake, is an important step for further processing. We then need a way to ensure that the data we have meets a given standard for quality — for example, are all of the values in their expected ranges? Are certain records missing important values? — and can impose these constraints while processing raw data from the data lake and storing structured data in a data warehouse. Given a source of clean, structured data, we can support exploratory analytics, report generation, and ultimately machine learning model training. In many cases, ad hoc queries and reports are related to the problems we might want to solve with machine learning: for example, quarterly reports will likely cover net loss or gain in subscribers and revenue, and a database programmer or business analyst might use interactive queries to identify attributes of customer records that are correlated with various business outcomes.

Data in our data warehouse is cleaned, federated, structured, organized, and (can be) labeled with various outcomes of interest for each customer — these are all properties that make it easier to find patterns and business value in the data. But relational databases are also typically organized with normal forms, which means that all of the relevant data about a given customer may be spread across multiple more-or-less independent tables. In order to prepare to train a machine learning model, we'll need to *denormalize* our data, and go from *long-form* tables of individual observations of a certain kind to *wide-form* tables whose rows include all of the relevant data we have for each customer.

Feature engineering

The wide-form data we have now is analogous to rows in a database table or programming-language objects — it's structured in a convenient format for further programmatic manipulation, but it's not in the most convenient format for training a machine learning model. The process of *feature engineering* is the process of building a technique to transform structured data into data that we can pass to a model training algorithm (or to a machine learning model). It entails identifying techniques to map from structured data to points in multidimensional space in such a way that the mapping preserves some interesting and meaningful structure of the source data; concretely, we might choose features, or attributes of each customer, and encode them as numbers so that we can encode customers as vectors of numbers in such a way that similar customers map to similar vectors and (ideally) there is a relatively straightforward way to partition the feature space between vectors for customers that churned and those for customers that have not yet churned. Once we've identified which features are important and how to encode them, we can develop a feature extraction pipeline that we can use to prepare structured data for model training or inference.

Model selection, training, tuning, and validation

At their core, machine learning models provide compact, useful summaries of datasets. In the case of the customer churn problem, our model will optimize a function to identify churning customers by identifying combinations of features that imply most strongly that a customer will churn. In order to train a model, we'll need to identify a modeling approach, tune the parameters (called hyperparameters) that govern its behavior, and evaluate its performance before ultimately validating that it has generalized by testing its performance on held-out (and thus novel) data.

Production deployment, monitoring, and feedback

Even a promising model can only be as useful as the system that employs it to solve a business problem. In order to deploy our model into production, we'll need

1. a way to reproduce the feature extraction pipeline and model training pipelines from our discovery workflow in a production environment as a production training pipeline (that is, a way to take raw, labeled data and produce a trained model), and
2. a way to publish our feature extraction pipeline and the trained model itself as a production inference pipeline (that is, a service that takes a record of novel data, extracts features from that record, and then uses the model to make a prediction about it)

In addition, we'll need to monitor our model's inputs and performance to account for *concept drift*, which occurs when the novel data we see have diverged enough from the data we trained our model on that its performance begins to materially suffer. (As a concrete example, a mobile network operator might start losing customers who spend a lot of time internationally roaming after a competitor introduces a free or reduced-cost international roaming plan – the underlying cause and phenomenon wouldn't have been captured in any training data collected before the competitive landscape changed.) Other changes that could impact the performance of our model include changes to upstream data formats or schemas, the introduction of new customer categories, and long-term trends in the overall market.

Identifying concept drift is one of many problems that can cause us to return to an earlier stage and revisit engineering and modeling decisions that we made. At the end of this workflow, we may have a model with excellent predictive performance that ultimately doesn't enable us to satisfy the right business metrics, requiring us to reevaluate how we formalized our problem. Concept drift may require retraining a model with additional training data. Unanticipated problems in production may suggest using a different overall approach. Finally, the processes of feature engineering and model training are often iterative since decisions made about how to encode features impact the kinds of models that we can effectively train.

1.2 Machine learning systems

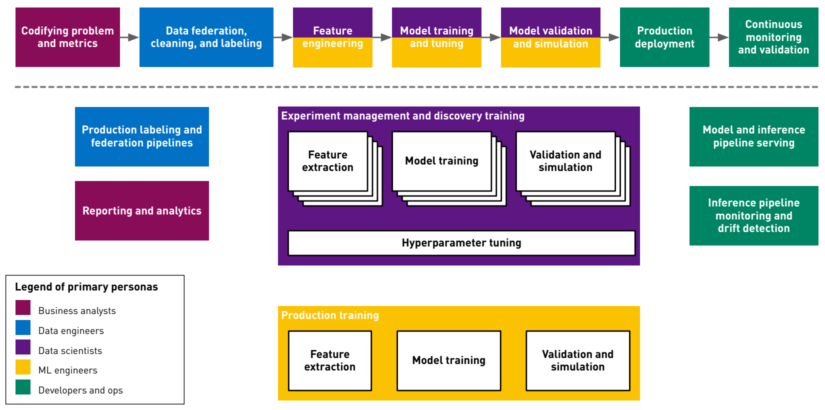

Codifying problem and metrics
Data federation, cleaning, and labeling
Feature engineering
Model training and tuning
Model validation and simulation
Production deployment
Continuous monitoring and validation
Production labeling and federation pipelines
Reporting and analytics
Experiment management and discovery training
Feature extraction
Model training
Validation and simulation
Hyperparameter tuning
Model and inference pipeline serving
Inference pipeline monitoring and drift detection
Legend of primary personas
Business analysts
Data engineers
Data scientists
ML engineers
Developers and ops
Production training
Feature extraction
Model training
Validation and simulation

Figure 1.2: A mapping from machine learning workflow stages to the machine learning system components each informs.

We've just discussed a human process by which teams develop machine learning systems, but we haven't explicitly described the machine learning systems they'll create. Each of the stages we discussed in the discovery workflow informs some parts of a machine learning system. There's a high-level view of one such mapping in the figure above, color-coded by the personas involved in each stage.

Some correspondences between human tasks and system components are obvious. For example, the feature engineering approach a data scientist chooses will directly inform the code that executes as part of a production feature extraction pipeline. The modeling approaches that are most successful in prototypes and experiments will inform the modeling approaches used in production, and so on.

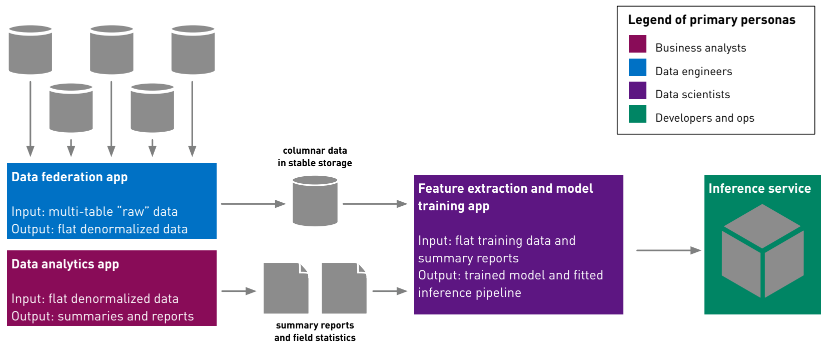
Some correspondences are more interesting. Just as the discovery workflow is iterative and lessons learned in later stages can feed back to earlier stages, explicit data feedback is an important part of machine learning systems: new (potentially-labeled) data collected during the operation of a system can feed back to the data federation pipeline to be used for future training efforts, metrics about model performance and business metrics can be tracked in a single dashboard (as well as monitored for evidence of concept drift), and experimental approaches can be evaluated in production in parallel to established ones.

Chapter 2

Overall application architecture

We'll be building up a complete system for churn modeling and prediction, consisting of four applications that work together:

- a data federation application that integrates structured data from a data warehouse,
- an analytics application that prepares human- and machine-readable reports from the federated data in order to generate business insight and support a data scientist's workflow,
- a feature extraction and model training application that takes flat training data and reports and generates a trained model and an inference pipeline, and
- an inference service that takes information about a customer and predicts whether or not that customer will churn.


Legend of primary personas
Business analysts
Data engineers
Data scientists
Developers and ops
Data federation app
Input: multi-table "raw" data
Output: flat denormalized data
Data analytics app
Input: flat denormalized data
Output: summaries and reports
columnar data in stable storage
summary reports and field statistics
Feature extraction and model training app
Input: flat training data and summary reports
Output: trained model and fitted inference pipeline
Inference service

As we've mentioned, we're going to focus on some of the parts of machine learning systems that are often ignored in machine learning technique tutorials: accelerating data federation, query processing, and exploratory analytics; managing the connections between different system components developed by different teams in different languages; and making the best use of our compute resources across the lifecycle of our system.

2.1 Technology stack

For our data federation and analytics applications, we'll be using Apache Spark and the RAPIDS Accelerator for Apache Spark, which enables us to accelerate Spark data frame operations on NVIDIA GPUs. Our feature extraction and model training application will use accelerated libraries from RAPIDS and the Python data ecosystem, including cuDF, cuML, Dask, and XGBoost. We'll use XGBoost and the RAPIDS Forest Inference Library to accelerate inference. There are some important differences in how these libraries work and how they achieve GPU acceleration; we'll briefly examine each.

RAPIDS

The RAPIDS libraries provide GPU-accelerated implementations of familiar interfaces from the Python data ecosystem. RAPIDS users are thus able to benefit from GPU acceleration without relying on implementing custom compute kernels or manually managing parallelism or transfers between host and device memory. cuDF is a GPU-accelerated *data frame* library, which allows users to manipulate collections of typed, structured data with an interface similar to the popular pandas library. cuML is a GPU-accelerated machine learning library that includes building blocks for feature extraction and model training pipelines that implement the scikit-learn estimator interface. cuML also includes a SHAP implementation for model explainability and FIL, an accelerated library for production inference with tree ensemble models. In addition, these libraries can share on-GPU data with other machine learning and deep learning libraries through the CUDA *array interface*. RAPIDS includes other libraries as well; the following figure shows how these map to the personas involved in the data science discovery lifecycle.

While cuDF and cuML provide familiar interfaces for data scientists, they are not drop-in replacements for pandas and scikit-learn respectively. Not every operation or algorithm in pandas and scikit-learn is amenable to GPU acceleration (or parallel execution in general); the RAPIDS libraries focus on implementing operations that can be accelerated on GPUs. A true drop-in replacement would need to transparently fall back to serial implementations for operations that won't parallelize; since it's possible, for example, to include serial scikit-learn estimators and transformers in a machine learning pipeline that uses cuML for performance-sensitive stages, it's possible for users to explicitly use serial implementations when necessary. In addition, most of the algorithms in RAPIDS are designed for a single node and a single GPU; in order to scale out or use more memory than is available in a single GPU, users will need to use the scale-out framework Dask in conjunction with cuDF and cuML, as in the following figure.

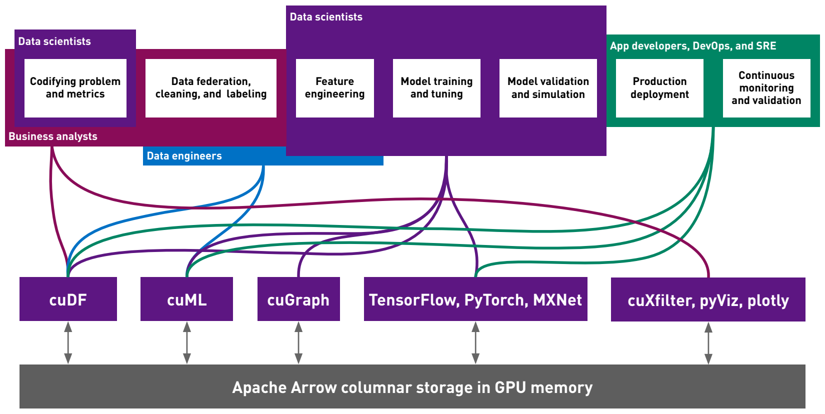

Data scientists
Codifying problem and metrics
Data federation, cleaning, and labeling
Data scientists
Feature engineering
Model training and tuning
Model validation and simulation
App developers, DevOps, and SRE
Production deployment
Continuous monitoring and validation
Business analysts
Data engineers
cuDF
cuML
cuGraph
TensorFlow, PyTorch, MXNet
cuXfilter, pyViz, plotly
Apache Arrow columnar storage in GPU memory

Figure 2.1: A mapping from personas and workflow stages to RAPIDS libraries, including cuDF, cuML, and cuGraph, and ecosystem projects that interoperate with these.

The RAPIDS Accelerator for Apache Spark

The RAPIDS Accelerator for Apache Spark takes a different approach to accelerating data science workloads on GPUs. Fundamentally, its approach is to provide *transparent acceleration* of Spark data frame jobs via a Spark plugin that integrates with Spark's query planner. The plugin rewrites data frame query plans in order to evaluate accelerable operations with implementations that use `libcudf` (the C++ library providing accelerated data frame functionality to the Python cuDF library) to execute on the GPU. Operations that cannot be accelerated will run on the CPU with Spark's built-in implementations; if there is a branch of a query plan that includes both accelerable and non-accelerable operations, the RAPIDS Accelerator plugin will automatically insert transfers between host and device memory so that both kinds of operations can work together transparently to execute a given query plan. The RAPIDS Accelerator for Apache Spark also provides an accelerated shuffle implementation using UCX (for data transfer within clusters) and integration with GPU-accelerated XGBoost.

Transparent acceleration has both a benefit and a cost for users. The benefit of transparent acceleration is that users can expect that a Spark application that runs successfully on the CPU will also run successfully with the RAPIDS Accelerator for Apache Spark enabled. The cost of transparent acceleration is that the performance improvement any given application can expect may be difficult to predict and will be a function of several factors: what percentage of its runtime it spends in accelerable data frame operations, how much work can be performed on the GPU between CPU-only operations, and how much each accelerable operation can improve when running on the GPU. Paradoxically, in order to take full advantage of transparent acceleration, data engineers and application developers may need to consider which parts of their application

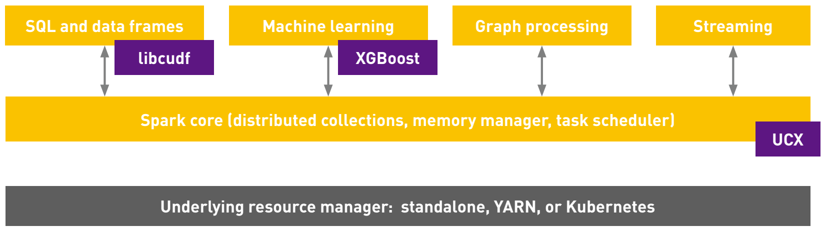
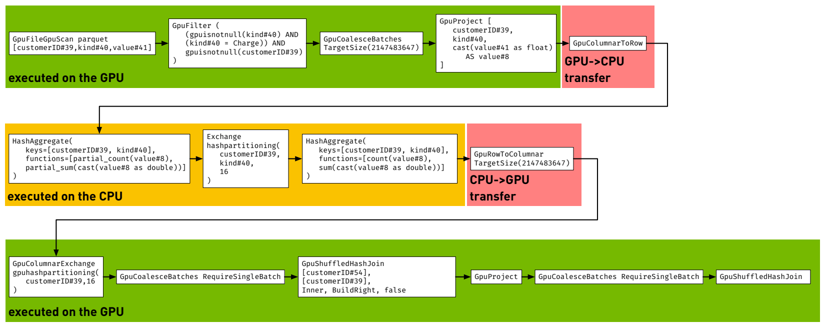

SQL and data frames
libcudf
Machine learning
XGBoost
Graph processing
Streaming
Spark core (distributed collections, memory manager, task scheduler)
UCX
Underlying resource manager: standalone, YARN, or Kubernetes

Figure 2.2: The architecture of GPU-acclerated Spark. Gray blocks in the bottom layer are the underlying resource managers Spark works with; the middle block is Spark's core task scheduler, resource manager, and low-level distributed collection API; the blocks on top are Spark's high-level APIs. Purple boxes indicate where extensions can provide GPU acceleration to Spark applications.

can be accelerated in order to make small changes for maximum performance. The following figure shows part of a query plan in which some operations are accelerated but one aggregate operation runs on the CPU.



```
GpuFileGpuScan parquet
[customerID#39,kind#40,value#41]

GpuFilter (
   (gpuisnotnull(kind#40) AND
   (kind#40 = Charge)) AND
   gpuisnotnull(customerID#39)
)

GpuCoalesceBatches
TargetSize(2147483647)

GpuProject [
   customerID#39,
   kind#40,
   cast(value#41 as float)
      AS value#8
]

GpuColumnarToRow

HashAggregate(
   keys=[customerID#39, kind#40],
   functions=[partial_count(value#8),
   partial_sum(cast(value#8 as double))]
)

Exchange
hashpartitioning(
   customerID#39,
   kind#40,
   16
)

HashAggregate(
   keys=[customerID#39, kind#40],
   functions=[count(value#8),
   sum(cast(value#8 as double))]
)

GpuRowToColumnar
TargetSize(2147483647)

GpuColumnarExchange
gpuhashpartitioning(
   customerID#39,16
)

GpuCoalesceBatches RequireSingleBatch

GpuShuffledHashJoin
[customerID#54],
[customerID#39],
Inner, BuildRight, false

GpuProject

GpuCoalesceBatches RequireSingleBatch

GpuShuffledHashJoin
```

Figure 2.3: A subset of an accelerated Spark query plan, showing GPU-accelerated portions, host-device and device-host transfers, and operations falling back to execute on the CPU.

Finally, transparent acceleration depends on applications using Spark's data frame and query abstractions. Because data frame operations are specified in an expressive but high-level API, it is possible to reorganize and rewrite query plans before they are executed, including replacing operations with higher-performance implementations (Spark itself takes advantage of this by generating native code to execute portions of query plans). Indeed, all data frame operations and SQL queries are planned and transformed by Spark before execution, and Spark itself provides a plugin layer so that external code (like the RAPIDS Accelerator) can alter the behavior of the query planner. Spark's lower-level resilient distributed dataset (RDD) API allows users to execute arbitrary code on partitioned distributed collections, but this additional flexibility of expression for application developers comes at the cost of flexibility of execution for Spark itself: since it is not, in general, feasible to safely transform arbitrary host-language code, Spark must treat the functions passed into its RDD API as opaque.

Chapter 3

Synthesizing data at scale

We'll start with where we get the data. A machine learning system is only as good as the data you build it with, so this is an important detail. However, since generating our synthetic data is not strictly a part of the final system, impatient readers are welcome to skip ahead to the next chapter where we describe the data federation app.

It's understandably difficult to find real-world customer data with which to demonstrate data processing and machine learning techniques. We began with an open-source synthetic dataset that is meant to be representative of a telecommunications company's customer records. Because this data set is open-source, there are several excellent extant tutorials using it as an example of feature engineering and model training techniques; some of these even involve deploying a model into production. However, all of these tutorials — like many data science tutorials — do not treat two important aspects of contemporary enterprise data pipelines:

1. They operate at minimal scale, since the source dataset is roughly 7,000 records. This is an excellent scale to let users experiment with modeling techniques quickly on modest hardware (like a laptop or tablet), but operating on a few thousand rows doesn't provide an opportunity to engage many of the challenges of processing larger datasets and training models at realistic scale.
2. They begin with denormalized, wide-form data as it might appear on a data scientist's desk, not with multiple tables from a data warehouse that need to be aggregated and federated in order to integrate all of the data we have about each given customer.

Our goal in this blueprint is to show an interesting churn modeling problem at meaningful scale, to show the query workloads that a data engineer might develop to prepare wide-form data for a data scientist, and to show exploratory analytic workloads that might support business intelligence applications. In order to do that, we'll need more data and normalized data. Here's how we'll get there:

1. We're going to augment the initial dataset by generating multiple customers that are essentially identical to each customer in the initial dataset, and
2. For each row in the augmented dataset, we're going to generate records in multiple tables corresponding to a normalized schema as it might exist in an enterprise data warehouse.

Specifically, the original dataset has the following attributes:

Name	**Type**	**Domain or notes**
`customerID`	string	
`Gender`	categorical	"Male" or "Female"
`SeniorCitizen`	boolean	
`Partner`	boolean	
`Dependents`	boolean	
`Tenure`	int	the age of the account in months
`PhoneService`	boolean	
`MultipleLines`	categorical	"yes," "no," or "no phone service"
`InternetService`	boolean	
`OnlineSecurity`	categorical	"yes," "no", or "no internet service"
`OnlineBackup`	categorical	"yes," "no", or "no internet service"
`DeviceProtection`	categorical	"yes," "no", or "no internet service"
`TechSupport`	categorical	"yes," "no", or "no internet service"
`StreamingTV`	categorical	"yes," "no", or "no internet service"
`StreamingMovies`	categorical	"yes," "no", or "no internet service"
`Contract`	categorical	"Month-to-month", "One year", or "Two year"
`PaperlessBilling`	boolean	
`PaymentMethod`	categorical	"Credit card (automatic)", "Mailed check", "Bank transfer (automatic)", "Electronic check"
`MonthlyCharges`	currency	a dollar amount
`TotalCharges`		currency
`Churn`	boolean	the label we're ultimately interested in predicting

When we generate synthetic data, we derive five related tables from the source data: **customer metadata** (basic information about our customer), **billing events** (account activations, terminations, and monthly charge events), **phone features** and **internet features** (long-form tables of pairs indicating that a given customer has enabled a given service on her account), and **account features** (specific features enabled on given customer accounts).

Some of the tables we're deriving contain *long-form* data, which is more convenient for manipulation and analysis but less convenient for model training than the *wide-form* data in the source dataset. (In database design terms, long-form data is *normalized*.) The next two tables show the difference between long- and wide-form data on four customers and three features. The first, long-form table has one observation in each row: a given customer for whom a given feature has a certain value. The second, wide-form table has several observations in each row: for each customer, a row contains all of the feature information we've collected. Note that the long-form table does not contain a row for every customer: customer `7469-LKBCI`, who has none of the relevant features, is not collected in the long-form table. This implies that we need a separate canonical list of customers outside of long-form feature tables.

Customer ID	**Feature**	**Value**
`7590-VHVEG`	InternetService	DSL
`6388-TABGU`	InternetService	DSL
`9237-HQITU`	InternetService	Fiber optic
`7590-VHVEG`	OnlineBackup	Yes
`6388-TABGU`	OnlineSecurity	Yes
`6388-TABGU`	OnlineBackup	Yes

Customer ID	**InternetService**	**OnlineSecurity**	**OnlineBackup**
`7590-VHVEG`	DSL	No	Yes
`9237-HQITU`	Fiber optic	No	No
`6388-TABGU`	DSL	Yes	Yes
`7469-LKBCI`	No	No service	No service

Chapter 4

Data federation

In this chapter, we'll examine the first part of our blueprint: a data engineering pipeline that federates and integrates structured transactional data and prepares it for further processing by a data scientist, with a special focus on ensuring our pipeline can take advantage of the RAPIDS Accelerator for Apache Spark.

4.1 Generating wide tables from customer account data

Our source data consists about five normalized tables about customer accounts and activity. Given these data sources, we can simulate the data engineering pipeline that would prepare a flat, wide table for a data scientist to process and model. Essentially, this consists of multiple joins, aggregations, and transformations, including

- Rolling up billing event counts (to calculate account tenure) and billing amounts (to calculate lifetime account value),
- Identifying whether an customer is a senior citizen or not based on their birthdate, and
- Reconstructing wide-form account features (services and billing data) from long-form tables

We've implemented this pipeline with a Python application that uses Apache Spark. Just to give you a flavor for the kind of application it is, the code that calculates account tenure and lifetime value looks like this:

```
counts_and_charges = billing_events.groupBy(
    "customerID", "kind"
).agg(
    F.count(billing_events.value).alias("event_counts"),
    F.sum(billing_events.value).alias("total_charges")
)
```

And the code that identifies churning customers looks like this:

```
terminations = billing_events.where(
    F.col("kind") == "AccountTermination"
).select(
    F.col("customerID").alias("Churn")
)

churned = customers.join(
    terminations,
    customers.customerID == terminations.Churn,
    how="leftouter"
).select(
    "customerID",
    F.when(F.col("Churn").isNull(),
    F.lit(False)).otherwise(F.lit(True)).alias("Churn")
)
```

In general, it's a pretty standard Spark application that reads structured data, processes some queries, aggregations, and joins, and then writes structured output.

4.2 Improving analytics performance

Remember that our overall data science workflow isn't a waterfall; it's a cycle. Since we might want to change the nature of our analytics job in order to meet new downstream requirements (or to fix bugs), we might have to rerun the job at any time. Since humans will often be waiting for the result of analytics jobs or ad hoc analytic queries, we want these jobs to be as fast as possible to make the most of human time and attention. Here are some techniques we used while developing the federation application; you can use these as well to improve the performance of your Spark query workloads.

Inspect

Use the tools that Spark provides to help understand what your job is actually doing: look at the web UI to determine where your application is spending time and ensure that it is adequately using all of the resources it has reserved. Inspect the query plans generated by `DataFrame.explain` to ensure that a seemingly-simple query doesn't imply a pathological execution plan.

Upgrade and optimize

There are often engineering costs involved with validating applications against new versions of frameworks, and enterprises can be conservative about upgrading Spark. But if you can run your application on Spark 3.0 or greater, you'll benefit from improved performance relative to the 2.x series, especially if you enable Adaptive Query Execution, which will use runtime statistics to dynamically choose better partition sizes, more efficient join types, and limit the impact of data skew.

Accelerate and understand

Once you're on Spark 3.0, consider using the RAPIDS Accelerator for Apache Spark, which has the potential to dramatically speed up DataFrame-based Spark applications. However, like any advanced optimization, you'll want to make sure you understand how to make the most of it. Here are some concrete tips:

- Some applications are better candidates for GPU acceleration than others. The RAPIDS Accelerator for Apache Spark works by translating query plans to run on columnar data in GPU memory and thus it cannot accelerate code that processes RDDs or operates on data one row at a time. Applications that spend a lot of time using RDDs as well as DataFrames, or that use complex user-defined functions, will not see the maximum possible speedup.
- Start with the suggested configuration in the tuning guide. Remember that if you want to use more than one GPU on a given system, you'll need to run more than one Spark worker process on that system — each JVM will only be able to access one GPU.
- Once you're up and running with the GPU, make sure to use the tools available to you again. This means revisiting the Spark web UI and `DataFrame.explain` — but this time, look out for query plan nodes that don't run on the GPU. You can also configure the plugin to tell you why certain parts of your application did or did not run on the GPU by setting the property `spark.rapids.sql.explain` to `ALL` or `NOT_ON_GPU`; if you set this option, make sure you have access to console output from your application.

Diving in a bit to understand how the application was executing on the GPU made it possible to unlock additional performance. When we looked at the query graph, we were able to determine that two parts of the query were unable to run on the GPU as we had configured the application: aggregating total lifetime account values and determining whether or not a customer with a given birthdate is a senior citizen. These two issues caused parts of the query to execute on CPU and parts to execute on the GPU, meaning that there were more transfers of data between the CPU and the GPU than strictly necessary; as it turned out, only the first one seriously impacted performance.

Fortunately, the fix was very simple: the RAPIDS Accelerator for Apache Spark will not perform aggregates on floating-point values by default because the roundoff behavior may be different from a CPU execution due to parallel execution.

However, we can accelerate these operations as well if we set the property `spark.rapids.sql.variableFloatAgg.enabled` to True. (If we were processing real money, we'd use a more precise number format, like arbitrary-precision decimals, but for modeling behavior, floating-point values are a great choice. We'll see some of the tradeoffs involved in using decimals later in the book.)

Chapter 5

Reporting and exploratory analytics

The first step in the data science discovery workflow is formalizing the problem we're trying to solve, which depends on understanding the data and understanding the business. A well-defined problem can help to codify the ways in which our analytics efforts ultimately provide business value (rather than merely achieving excellent model performance metrics). Exploratory analysis can support formalizing the problem, developing these necessary understandings, and more:

1. Understanding the business impact of an effective solution is important for prioritizing efforts
2. Business context helps practitioners identify finer-grained success criteria.
3. Data scientists and business analysts need to define meaningful prediction targets for the models they'll ultimately be training
4. Better understanding of the data can inspire novel modeling approaches.
5. Exploratory analysis can support the data science workflow's "inner loop" of feature extraction, model training, and validation and simulation

Prioritizing efforts to maximize business impact

Understanding the business impact of an effective solution is important for *prioritizing efforts* (not to mention that individual data scientists, like other employees, are interested in avenues to demonstrate the quantifiable impact of their work). While data scientists may take great pride in producing robust, general models with minimal prediction error, the business impact — not merely the predictive power of their models — should guide their efforts, and exploratory analysis is an important tool to identify the extent to which improved predictive performance might affect business metrics.

Tightening success criteria with business context

Business context helps practitioners *identify finer-grained success criteria*. While any organization wants to treat every customer equally in providing excellent service, it may not want to treat the risk of churn for every customer as equally important to the business. We may thus be interested in ranking churn risks by their expected remaining lifetime account value, by the expected cost to acquire a comparable account, or by other metrics.

Defining meaningful prediction targets

Data scientists and business analysts need to *define meaningful prediction targets* for the models they'll ultimately be training. On a long enough timeline, every customer will fail to renew their subscription, but a model that asserts "yes, eventually" for every customer isn't useful or actionable. A model that exclusively identifies that customers who have recently begun the process to close their accounts are likely to churn is similarly dubious. Exploratory analysis can inform a carefully-crafted prediction target by enabling data scientists and analysts to simulate the plausibility and impact of various prediction targets on historical data.

Inspiring novel modeling approaches

Better understanding of the data can *inspire novel modeling approaches*. For example, two customer attributes may not be strongly correlated with churning individually but their combination may be. Exploratory analysis can thus inform the process of feature engineering.

Supporting discovery workflows with ubiquitous exploratory analysis

Exploratory analysis can *support the data science workflow's "inner loop"* of feature extraction, model training and tuning, and validation and simulation in two ways: directly, by providing summary statistics, domains for categorical features, and distributions for numerical features, and indirectly, by enabling data scientists to disregard uninformative features before training a model and thus making the model and the system built around it more robust.

5.1 Business analytics and reporting

While exploratory analysis is a valuable part of the early stages of the data science discovery workflow, similar workloads are important in production and *can provide insight and value even without training a model*. The main difference between these workloads is one of context: while exploratory analysis is generally ad hoc, business analytics workloads are typically run regularly in production. Techniques or queries used in exploratory analysis may inform or even become parts of automated reporting or analytics workloads.

5.2 Analytics in churn modeling

In our blueprint application, we'll incorporate a pair of analytics workloads. The first workload produces a machine-readable summary report that is — along with the single wide table of customer data we described earlier — part of the input to the model training application. The second workload produces a series of reports intended to help analysts and stakeholders better understand the factors that make customers more likely to renew or churn, in order to guide human decisions in the modeling process as well as to inform business decisions and service offerings that incorporate effective renewal incentives. We'll now examine each of these in turn.

Calculating feature summaries and domains

Part of understanding our data is understanding each feature individually; this is also a prerequisite to effective feature engineering or model training. Basic summary statistics — like minimum, count, mean, median, variance, and so on — are useful but may be insufficient to characterize a dataset alone, since radically different datasets may have similar summary statistics. A famous example of this phenomenon is Anscombe's Quartet, which is four two-variable datasets that have identical means, variances, correlations, and linear relationships but which exhibit obviously different shapes when plotted.

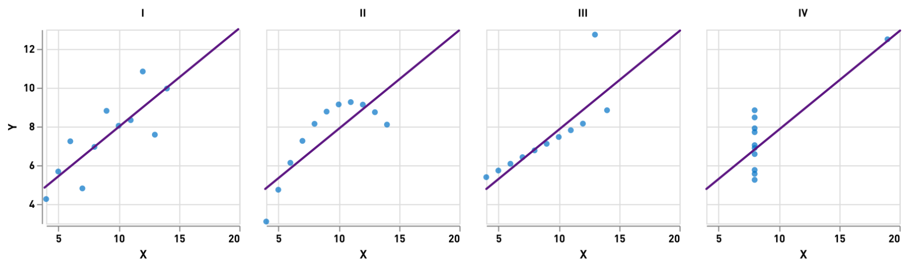


Figure 5.1: Anscombe's Quartet, a collection of synthetic datasets that have identical summary statistics but different shapes.

In order to more faithfully characterize our datasets, we'll need to compute more descriptive summaries of individual features in addition to the basic descriptive statistics. One such summary is the *cumulative distribution*, which can both inform feature engineering decisions and provide valuable business context. For example, in many cases, it is more useful to know that a given customer's monthly spend is in the 97th percentile than to know that that customer's monthly spend is two standard deviations above the mean. Apache Spark supports efficient techniques for calculating approximate quantiles of columns in data frames, which we can use to produce cumulative distributions of these values.

We also produce a report including basic summary statistics, such as mean, minimum, maximum, and variance (for numeric features) and the identities and counts of distinct values (for discrete or categorical columns). These statistics and distributions are useful in themselves but will also make the data scientist's job easier, since they can inform feature encoding approaches.

5.3 Exploratory analysis and reporting in churn modeling

The second component of our analytics workload simulates both exploratory analysis and scheduled reporting by producing two kinds of reports:

1. A set of *data cubes*, or multidimensional spreadsheets, showing the counts of customers with combinations of given attributes that churned or renewed, and
2. A collection of *rollup reports* showing the total lifetime account value of every customer that churned in a given quarter.

The data cube reports enable analysts to quickly drill down on a combination of features and see which are most strongly correlated with renewal or churn. It is worth noting that many real-world analytics pipelines may end at this point, since reports like this can provide actionable insight even without a trained predictive model. The next two figures show the value of these reports: by drilling down to plot the interaction of a customer's contract status (month-to-month, annual, or two-year) and their tenure in quarters, we can clearly establish that most of the customers who churn are relatively new and on month-to-month contracts.

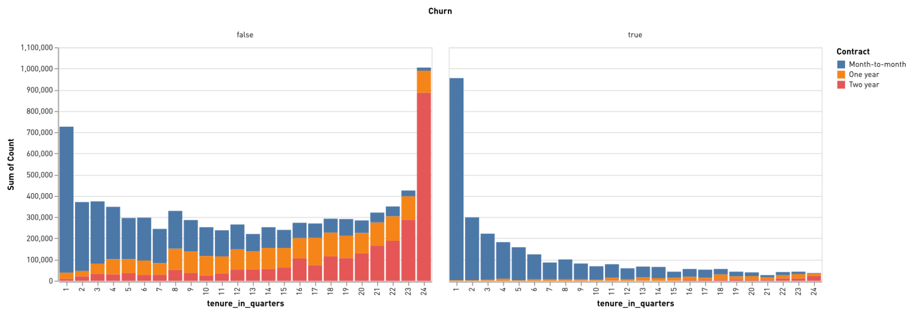


Figure 5.2: The distributions of a customer's tenure and contract status for customers who renewed (on the left) and who churned (on the right), demonstrating that new customers on month-to-month contracts represent the majority of churning customers.

These figures show another advantage of exploratory analysis and reporting: since some of the features we consider in our summaries are numeric

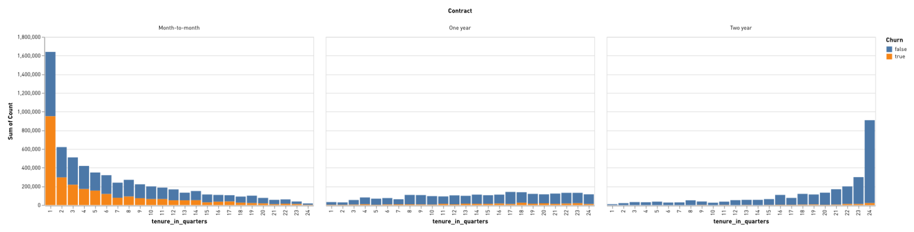


Figure 5.3: The distribution of renewing (blue) and churning (orange) customers by tenure in quarters, grouped by contract type, from least restrictive (on the left) to most restrictive (on the right).

(e.g., months of tenure), we can also use data cube reports to investigate the impact of various quantization or bucketing strategies. Instead of seeing how many customers churned at each discrete month of tenure, for example, we might be more interested in abstracting the tenure of our customers by looking at how many quarters, half-years, or years an account had been active. (In our plots, we bucketed customer tenures by quarter.) Identifying how best to abstract numeric data in order to convey the most information to human stakeholders and to downstream analyses alike is another goal of exploratory analysis.

5.4 Performance improvements and future work

In the first chapter, we saw how the RAPIDS Accelerator for Apache Spark could execute data federation workloads on NVIDIA GPUs. While some organizations may execute federation pipelines regularly, others may execute federation pipelines only when the pipelines themselves or the source data materially change. Exploratory analytics and reporting are more performance-sensitive, though: exploratory analytics workloads are typically interactive and human time is precious, and reporting workloads may be batch scheduled but run regularly. These workloads are also typically more complex than data federation workloads and thus also more amenable to performance improvement. By executing our analytics workloads on NVIDIA GPUs with the RAPIDS Accelerator for Apache Spark, we were able to achieve speedups of **nearly 700%** relative to CPU execution.

Chapter 6

Best practices for accelerated analytics

In this chapter, we finish presenting the analytics and federation components of our application and explain some best practices for getting the most out of Apache Spark and the RAPIDS Accelerator for Apache Spark.

6.1 Architecture review

Recall that our blueprint application includes a federation workload and a pair of analytics workloads.

1. The **federation workload** produces a single denormalized wide table of data about each customer drawn from aggregating data spread across five normalized tables of observations related to different aspects of customers' accounts.
2. The **first analytic workload** produces a machine-readable summary report of value distributions and domains for each feature.
3. The **second analytic workload** produces a series of illustrative business reports about customer outcomes.

We've implemented these three workloads as a single Spark application with multiple phases:

1. The app federates raw data from multiple tables in HDFS (which are stored as Parquet files) into a single wide table.
2. Because the wide table is substantially smaller than the raw data, the app then reformats the wide output by coalescing to fewer partitions and casting numeric values to types that will be suitable for ML model training. The output of this phase is the source data for ML model training.

3. The app then runs the analytics workloads against the coalesced and transformed wide table, first producing the machine-readable summary report and then producing a collection of rollup and data cube reports.

6.2 Performance considerations

Parallel execution

For over 50 years, one of the most important considerations for high performance in computer systems has been increasing the applicability of parallel execution. (We choose, somewhat arbitrarily, to identify the development of Tomasulo's algorithm in 1967, which set the stage for ubiquitous superscalar processing, as the point at which concerns about parallelism became practical and not merely theoretical.) In the daily work of analysts, data scientists, data and ML engineers, and application developers, concerns about parallelism often manifest in one of a few ways; we'll look at those now.

When scaling out, perform work on a cluster

If you're using a scale-out framework, perform work on a cluster instead of on a single node whenever possible. In the case of Spark, this means executing code in Spark jobs on executors rather than in serial code on the driver. In general, using Spark's API rather than host-language code in the driver will get you most of the way there, but you'll want to ensure that the Spark APIs you're using are actually executing in parallel on executors.

Operate on collections, not elements; on columns, not rows

A general best practice to exploit parallelism and improve performance is to use specialized libraries that perform operations on a collection at a time rather than an element at a time. In the case of Spark, this means using data frames and columnar operations rather than iterating over records in partitions of RDDs; in the case of the Python data ecosystem and RAPIDS, it means using vectorized operations that operate on entire arrays and matrices in a single library call rather than using explicit looping in Python. Crucially, both of these approaches are also amenable to GPU acceleration.

Amortize the cost of I/O and data loading

I/O and data loading are expensive, so it makes sense to amortize their cost across as many parallel operations as possible. We can improve performance both by directly reducing the cost of data transfers and by doing as much as possible with data once it is loaded. In Spark, this means using columnar formats, filtering relations only once upon import from stable storage, and performing as much work as possible between I/O or shuffle operations.

Better performance through abstraction

In general, raising the level of abstraction that analysts and developers employ in apps, queries, and reports allows runtimes and frameworks to find opportunities for parallel execution that developers didn't (or couldn't) anticipate.

Use Spark's data frames

As an example, there are many benefits to using data frames in Spark and primarily developing against the high-level data frame API, including faster execution, semantics-preserving optimization of queries, reduced demand on storage and I/O, and dramatically improved memory footprint relative to using RDD based code. But beyond even these benefits lies a deeper advantage: because the data frame interface is high-level and because Spark allows plug-ins to alter the behavior of the query optimizer, it is possible for the RAPIDS Accelerator for Apache Spark to replace certain data frame operations with equivalent — but substantially faster — operations running on the GPU.

Transparently accelerate Spark queries

Replacing some of the functionality of Spark's query planner with a plug-in is a particularly compelling example of the power of abstraction: an application written years before it was possible to run Spark queries on GPUs could nevertheless take advantage of GPU acceleration by running it with Spark 3.1 and the RAPIDS Accelerator.

Maintain clear abstractions

While the potential to accelerate unmodified applications with new runtimes is a major advantage of developing against high-level abstractions, in practice, maintaining clear abstractions is rarely a higher priority for development teams than shipping working projects on time. For multiple reasons, details underlying abstractions often leak into production code; while this can introduce technical debt and have myriad engineering consequences, it can also limit the applicability of advanced runtimes to optimize programs that use abstractions cleanly.

Consider operations suitable for GPU acceleration

In order to get the most out of Spark in general, it makes sense to pay down technical debt in applications that work around Spark's data frame abstraction (e.g., by implementing parts of queries as RDD operations). In order to make the most of advanced infrastructure, though, it often makes sense to consider details about the execution environment without breaking abstractions. To get the best possible performance from NVIDIA GPUs and the RAPIDS Accelerator for Apache Spark, start by ensuring that your code doesn't work around abstractions, but then consider the types and operations that are more or less amenable to GPU execution so you can ensure that as much of your applications run on the GPU as possible. We'll see some examples of these next.

Types and operations

Not every operation can be accelerated by the GPU. When in doubt, it always makes sense to run your job with `spark.rapids.sql.explain` set to `NOT_ON_GPU` and examine the explanations logged to standard output. In this section, we'll call out a few common pitfalls, including decimal arithmetic and operations that require configuration for support.

Beware of decimal arithmetic

Decimal computer arithmetic supports precise operations up to a given precision limit, can avoid and detect overflow, and rounds numbers as humans would while performing pencil-and-paper calculations. While decimal arithmetic is an important part of many data processing systems (especially for financial data), it presents a particular challenge for analytics systems. In order to avoid overflow, the results of decimal operations must widen to include every possible result; in cases in which the result would be wider than a system-specific limit, the system must detect overflow. In the case of Spark on CPUs, this involves delegating operations to the `BigDecimal` class in the Java standard library and precision is limited to 38 decimal digits, or 128 bits. The RAPIDS Accelerator for Apache Spark can currently accelerate calculations on decimal values of up to 18 digits, or 64 bits.

We've evaluated two configurations of the churn blueprint: one using floating-point values for currency amounts and one using decimal values for currency amounts (which is the configuration that the performance numbers we're currently reporting is running against). Because of its semantics and robustness, decimal arithmetic is more costly than floating-point arithmetic, but it can be accelerated by the RAPIDS Accelerator plugin as long as all of the decimal types involved fit within 64 bits.

Configure the RAPIDS Accelerator to enable more operations

The RAPIDS Accelerator is conservative about executing operations on the GPU that might exhibit poor performance or return slightly different results than their CPU-based counterparts. As a consequence, some operations that could be accelerated may not be accelerated by default, and many real-world applications will need to enable these to see the best possible performance. We saw an example earlier, in which we had to explicitly enable floating-point aggregate operations in our Spark configuration by setting `spark.rapids.sql.variableFloatAgg.enabled` to `true`. Similarly, when we configured the workload to use decimal arithmetic, we needed to enable decimal acceleration by setting `spark.rapids.sql.decimalType.enabled` to `true`.

The plugin documentation lists operations that can be supported or not by configuration and the reasons why certain operations are enabled or disabled by default. In addition to floating-point aggregation and decimal support, there are several classes of operations that production Spark workloads are extremely likely to benefit from enabling:

- cast operations, especially from string to date or numeric types or from floating-point types to decimal types.
- string uppercase and lowercase (e.g., `SELECT UPPER(name) FROM EMPLOYEES`) are not supported for some Unicode characters in which changing the case also changes the character width in bytes, but many applications do not use such characters. You can enable these operations individually or enable them and several others by setting `spark.rapids.sql.incompatibleOps.enabled` to tr‘ue.
- reading specific types from CSV files; while reading CSV files is currently enabled by default in the plugin (`spark.rapids.sql.format.csv.enabled`), reading invalid values of some types (numeric types, dates, and decimals in particular) will have different behavior on the GPU and the CPU and thus reading each of these will need to be enabled individually.

Accelerate data ingest from CSV files

CSV reading warrants additional attention: it is expensive and accelerating it can improve the performance of many jobs. However, because the behavior of CSV reading under the RAPIDS Accelerator may diverge from Spark's behavior while executing on CPUs and because of the huge dynamic range of real-world CSV file quality, it is particularly important to validate the results of reading CSV files on the GPU. One quick but valuable sanity check is to ensure that reading a CSV file on the GPU returns the same number of NULL values as reading the same file on the CPU. Of course, there are many benefits to using a self-documenting structured input format like Parquet or ORC instead of CSV if possible.

Avoid unintended consequences of query optimization

The RAPIDS Accelerator transforms a physical query plan to delegate certain operators to the GPU. By the time Spark has generated a physical plan, though, it has already performed several transformations on the logical plan, which may involve reordering operations. As a consequence, an operation near the end of a query or data frame operation as it was stated by the developer or analyst may get moved from a leaf of the query plan towards the root.

In general, this sort of transformation can improve performance. As an example, consider a query that, as written, joins two data frames on the values of a column and then filters the results based on a selective predicate. Executing this query as written, as in Figure 2, will likely generate a large relation

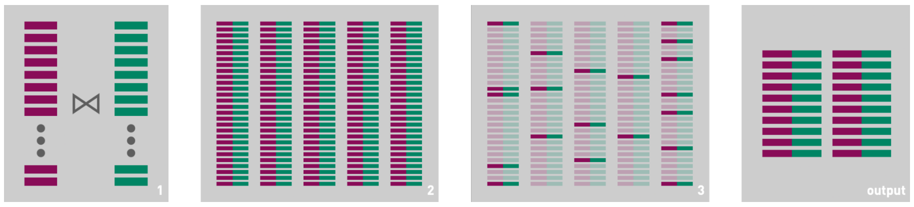


Figure 6.1: A depiction of executing a data frame query that joins two data frames and then filters the results. If the predicate is sufficiently selective, most of the output tuples will be discarded.

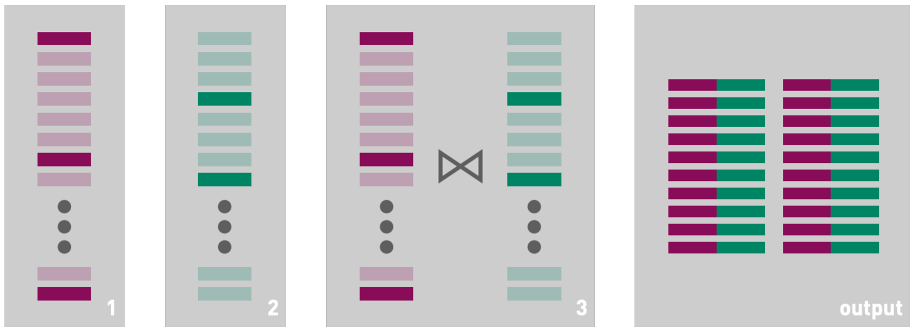


Figure 6.2: A depiction of executing a data frame query that filters two input relations before joining the results. If the predicate can be evaluated on each input relation independently, this query execution produces the same results as the query execution in the previous figure, but much more efficiently.

from the join and discard most of the results. When possible, it will often be more efficient to execute the filter before executing the join, as in Figure 3. Doing so will reduce the cardinality of the join, eliminate comparisons that will ultimately be unnecessary, decrease memory pressure, and potentially even reduce the number of data frame partitions that need to be considered in the join.

However, these optimizations can have counterintuitive consequences and aggressive query reordering may negatively impact performance on the GPU if the operation that is moved towards the root of the query plan is only supported on CPU or if it generates a value of a type that is not supported on the GPU. When this happens, a greater percentage of the query plan may execute on the CPU than is strictly necessary. You can often work around this problem and improve performance by dividing a query into two parts that execute separately, thus forcing CPU-only operation near the leaves of a query plan to execute only after the accelerable parts of the original query run on the GPU.

Chapter 7

Interoperability considerations

Our focus now shifts from analytics and data engineering to machine learning. Production machine learning systems are typically designed, developed, and maintained by cross-functional teams, including data engineers, business analysts, data scientists, machine learning engineers, application developers, and devops and site reliability engineers. As we saw in the introduction, different individuals and teams are responsible for different components at different stages of a system's lifecycle. In this chapter, we'll cover some of the concerns involved in sharing data between data engineering and analytics teams and data science teams.

7.1 Preserve performance while sharing Parquet datasets across ecosystems

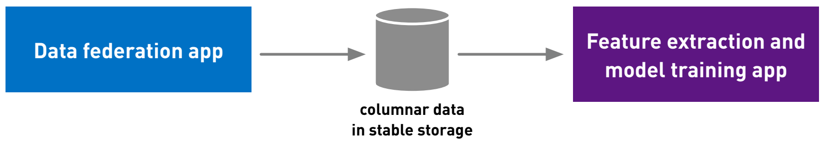
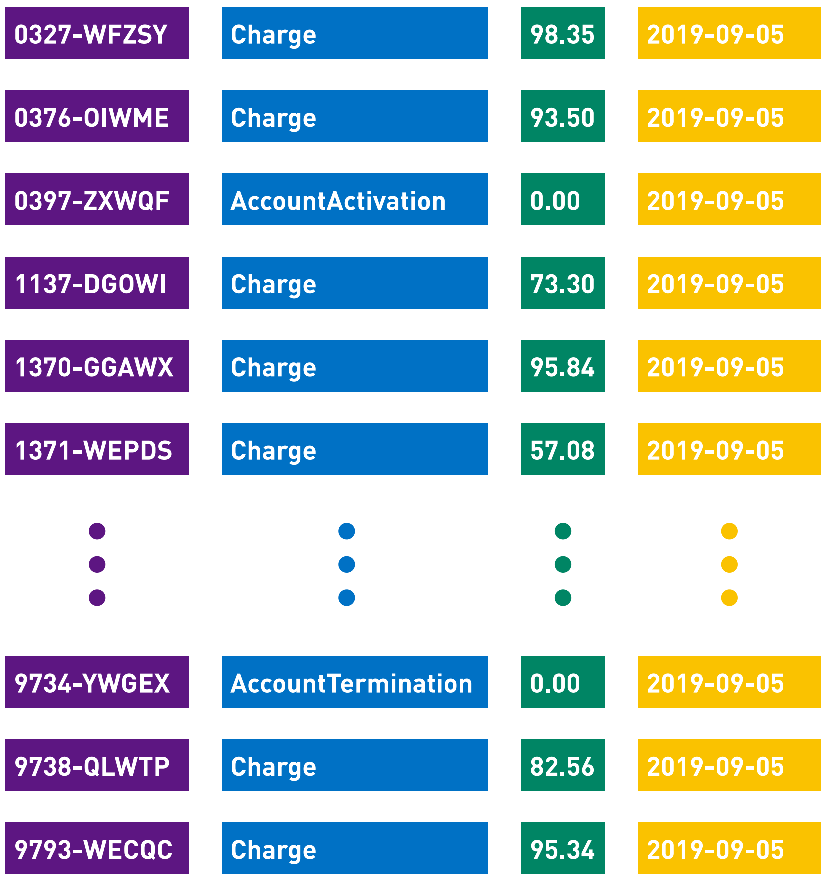


Figure 7.1: A detail of our blueprint architecture, focusing on the handoff point between data federation and model training applications.

The interfaces between software components developed by different teams are an important part of how the humans in these teams collaborate. In our application, the interface between data federation and analytics and machine learning is federated wide-table data in an Apache Parquet file. Parquet is an ubiquitous columnar format that enjoys excellent support in both the (primarily JVM-based and data engineering-focused) big data ecosystem and the (primarily Python-based) machine learning and experimental data science ecosystem. However, that ubiquity can mask some challenges in sharing data between teams and across environments. In order to understand those challenges, let's briefly review how Apache Parquet stores data.

Understanding Parquet

Apache Parquet is a *columnar format* for structured data, meaning that data are stored so that values in each column are physically adjacent (rather than so that each value in a given row is physically adjacent). We might imagine an excerpt from our billing events data as laid out in an abstract table representation like this:



0327-WFZSY	Charge	98.35	2019-09-05
0376-OIWME	Charge	93.50	2019-09-05
0397-ZXWQF	AccountActivation	0.00	2019-09-05
1137-DGOWI	Charge	73.30	2019-09-05
1370-GGAWX	Charge	95.84	2019-09-05
1371-WEPDS	Charge	57.08	2019-09-05
⋮	⋮	⋮	⋮
9734-YWGEX	AccountTermination	0.00	2019-09-05
9738-QLWTP	Charge	82.56	2019-09-05
9793-WECQC	Charge	95.34	2019-09-05

Figure 7.2: An abstract representation of tabular data, with records in rows and fields in columns.

A *row-oriented representation* of these data would store each record contiguously in a file, so that all of the field values for a given row are adjacent on disk, as in the following figure (the first record is highlighted):

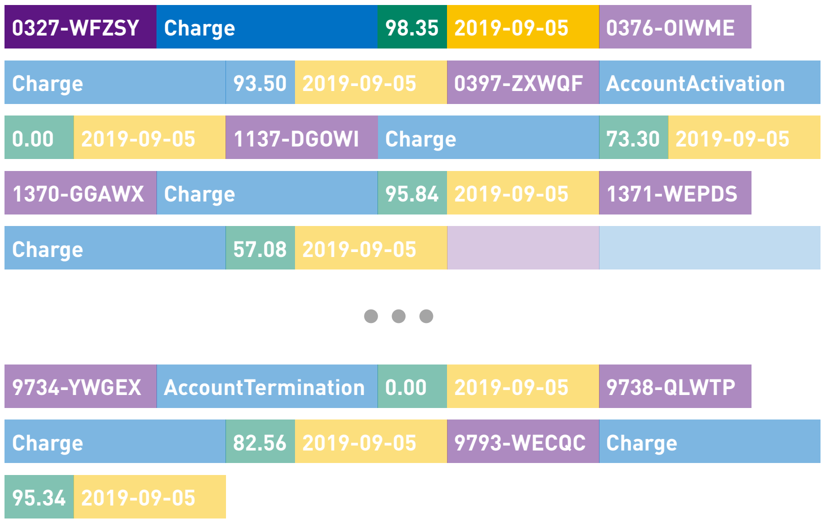


Figure 7.3: A portion of the billing events table, represented in a row-oriented storage format, with field values from the same record adjacent to one another

A *column-oriented representation*, however, stores all of the values for each field contiguously, as in the following figure (the first record is highlighted again):

Columnar representations are often more efficient for analytics use cases, in which it is more likely to operate on every value in a column (or field) than every value in a row (or record), and in which several fields in a record may be projected out and may not contribute to a result.[1] Because the values for a given field in consecutive records are adjacent on disk and in memory, operating on a given field value for every record involves reading from disk or memory sequentially and can be accelerated by caching and prefetching at multiple layers of the memory hierarchy. By contrast, reading a single field from multiple records in a row-oriented representation will require inefficient seeking and pollute system and hardware caches with unused data.

However, columnar organization enables additional optimizations that can further improve analytic efficiency.

Columns with relatively few values can be *dictionary-encoded*, so that each individual value is only stored once, in a dictionary in file metadata, and each field value on disk is replaced with its index in the dictionary. For example, the `kind` field of our billing events table can take on three values: `AccountActivation`, `AccountTermination`, and `Charge`. By storing each value once, we'd create a dictionary like this:

[1]Row-oriented representations are often more appropriate for transaction processing than for analytic processing. In general, because of how they lay out data, they can be more efficient if every value in given record will be read or written in a given transaction or if only a very small subset of records will be accessed to compute a query result.

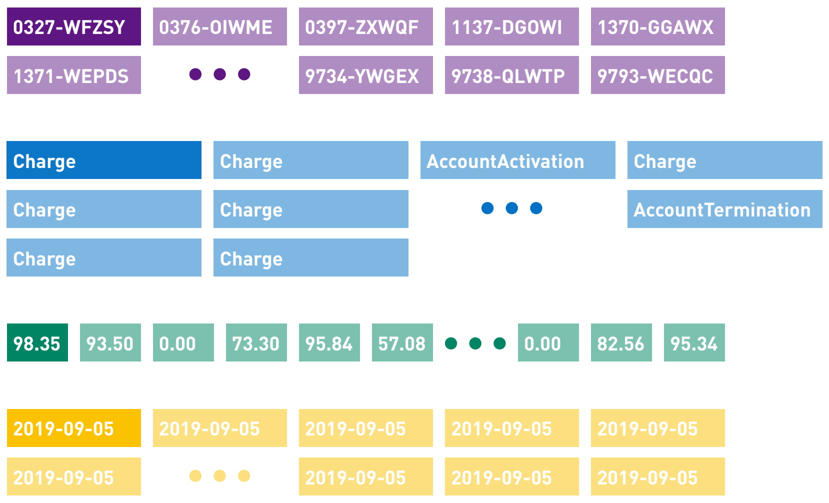


Figure 7.4: A portion of the billing events table, represented in a column-oriented storage format, with a separate logical file for the values in each column

```
{0: "AccountActivation", 1: "AccountTermination", 2: "Charge"}
```

and then replace every value with its index in the dictionary: 0, 1, or 2.

Columns in which consecutive records might share the same or similar values can be *run-length encoded*. Instead of storing identical values multiple times consecutively, it's possible to store the value once along with a representation of the number of times it repeats; this can be extremely efficient for values like timestamps since many consecutive events can occur on the same day (or, in some systems, even in the same second).

These kinds of optimizations can be combined to save even more space in a data representation. The next figure first shows using dictionary encoding (on the left) and then dictionary encoding and run-length encoding (on the right) to compress the `kind` column of the billing events table.

Columnar formats in general (and Parquet in particular) enable many other additional optimizations, such as *predicate pushdown*, in which a file or portion of a file may be ignored altogether if its metadata indicate that it does not contain any values that will be relevant to computing a query (for example, if we're querying for billing events from September 2020, we needn't examine a file whose most recent event was from January 2019). While a full discussion of Parquet's capabilities and implementation is beyond the scope for this book, it should be clear that the kinds of optimizations it enables greatly impact performance in several ways:

- by improving I/O throughput (by reducing the size of datasets and storing values likely to be accessed together sequentially),

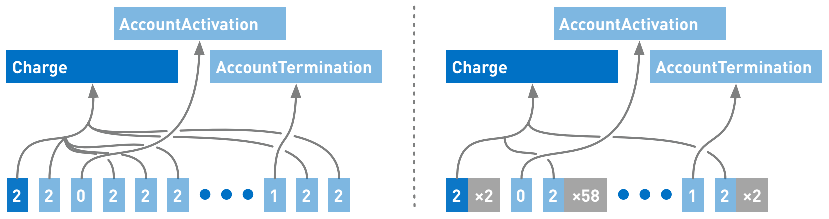


Figure 7.5: Example optimizations enabled by column-oriented storage formats: on the left, the `kind` column from the billing events table is dictionary-encoded; on the right, the dictionary-encoded data are also run-length encoded

- by reducing memory consumption (by storing data more compactly and with a locality-sensitive format in memory), and
- by eliminating unnecessary computation (after exploiting redundancy and metadata to identify unnecessary operations).

We are interested in preserving these performance advantages when we use Parquet to share data between our data engineers (and our federation and analytics applications) and our data scientists (and our model training application). We'll now examine where this can go wrong.

Maintain categorical encoding when loading Parquet files in pandas

Apache Parquet is well-supported both in the JVM-based big data ecosystem (through the Parquet project itself and its integrations in Hadoop-ecosystem projects including Spark) and in the Python data ecosystem (via functionality in Apache Arrow and other libraries). However, some projects rely on informal conventions that are not part of the Parquet specification. As a consequence, certain optimizations and features may not survive the round trip between serialization in one environment and deserialization in another.

For example, both Apache Spark and pandas write Parquet files with implementation-specific metadata in order to map from Parquet types to Spark data frame types or pandas types respectively. Spark and pandas each know how to read their own metadata, but other consumers may not without special configuration. In particular, pandas will use dictionary encoding to efficiently store *categorical features*, or columns that may assume only one of a finite (and typically relatively small) set of values.[2] When pandas saves a data frame as a Parquet file, the Parquet file includes a schema that uses Parquet-native types and encodings to store categoricals and also pandas-specific metadata to indicate that these values should be interpreted

[2]Dictionary encoding is a particular implementation of *label encoding*, in which unique values in some domain are mapped to unique small integers; it is a relatively standard part of many approaches for statistical and machine learning software to handle categoricals even without involving Parquet.

as categoricals on load and thus stored efficiently in a pandas data frame in memory.

Because pandas reads its metadata when loading a Parquet file that has been saved from pandas, it is able to efficiently load categorical features from dictionary-encoded string columns. However, when pandas loads a Parquet file generated by Apache Spark, the pandas-specific type metadata is not present, meaning that categoricals stored as dictionary-encoded string columns may be loaded as strings, resulting in substantial memory and computational overhead. Concretely, if we were to load the raw billing events table, which was generated with Spark, into a pandas data frame, the code would look like this:

```
import pandas as pd
billing_events = pd.read_parquet("billing_events.parquet")
```

If we were to inspect the pandas schema for this data frame (by inspecting the `dtypes` attribute of `billing_events`), we'd see that the `kind` field, which is a categorical, has been materialized as strings and stored in memory as Python objects. We can label-encode this column or cast it to a categorical, but we shouldn't have to — and the memory overhead of the

Fortunately, it's possible for pandas to efficiently consume the Parquet files that Spark produces, if we take a little care to exploit the available metadata. pandas itself doesn't include a Parquet reader implementation, but its `read_parquet` function is able to use either Apache Arrow (through the `pyarrow` library) or the `fastparquet` library if they are available. In both cases, the `read_parquet` function in pandas can pass additional options to the underlying Parquet implementation. In our blueprint, we're using Arrow, and in order to take advantage of Spark-generated metadata when loading files into pandas, we'll need to tell Arrow how to handle particular fields. We'll do this by passing the `read_dictionary` keyword argument to `read_parquet`, which will in turn be passed to Arrow.

```
import pandas as pd
billing_events = pd.read_parquet("billing_events.parquet",
                                 read_dictionary=["kind"])
```

If we do this, pandas will correctly interpret the dictionary-encoded fields as categoricals. In your own applications, you'll know which fields represent categoricals and should be read as such; doing so will preserve performance and eliminate potential integration headaches.

Double-check performance and correctness when using multiple Parquet implementations

Materializing dictionary-encoded strings instead of representing them as categoricals is a good example of the sort of integration pitfall stemming from using multiple Parquet implementations, because it can result in a particularly egregious performance penalty. However, there are other interoperability concerns to watch out for when using Parquet files as an interface between application components implemented in different language ecosystems. In this section, we'll briefly examine some potential issues that you might encounter in your own systems.

Avoid structure-valued columns. One of the advantages of Parquet in the big data ecosystem is its efficient support for structured data, including deeply-nested data. While Arrow itself can load structured data (with the `read_parquet` function in the `pyarrow.parquet` module), structure-valued fields will turn into Python `dict` objects once they're converted into a pandas data frame, which provides an inefficient implementation of an inconvenient interface. When using Parquet as an interface between components developed in different ecosystems, prefer flat schemas.

Identify compatible compression algorithms. In addition to encoding values within columns to save space, Parquet can also compress individual column chunks with a general-purpose compression algorithm. The current version of the Parquet specification supports several different compression algorithms; `GZIP`, `SNAPPY`, and `BROTLI` are the most widely supported. (While it is possible to store encoded column data without compression, using compression will often improve overall system performance.) If you know in advance that every component of your application uses a Parquet implementation that supports another compression algorithm, you can use that algorithm; otherwise, it's safest to stick to `GZIP`, `SNAPPY`, or `BROTLI`.

Pay careful attention to the behavior of alternative data frame implementations. In complex accelerated pipelines, data science teams will often be working not merely with pandas but also with cuDF and Dask data frames. These alternative implementations may have different behavior than pandas, especially in corner cases, so it's important to verify that they behave as you expect they will. As a concrete example, consider the categorical case we presented earlier: if we're using Dask with cuDF, we'll need to use the `categorize()` method to turn objects into categoricals, like this:

```
import dask_cudf as dc
ddf = dc.read_parquet("billing_events.parquet").categorize(["kind"])
```

Dask will ultimately compute the possible values from each categorical across every partition, although it may be able to make use of Dask-specific metadata (so it may make sense for a data scientist to write out intermediate files from Dask, which will have this metadata).

If we're using cuDF on a single GPU, we can get the proper behavior by first reading our dataset into a pandas data frame and then converting it to a cuDF data frame:

```
import pandas as pd
import cudf
```

```
pd_be = pd.read_parquet("billing_events.parquet",
                        read_dictionary=["kind"])

billing_events = cudf.DataFrame(pd_be)
```

(If we do not need to preserve categoricals, we can use cuDF's native Parquet reader, which should be faster.)

7.2 Summary

Parquet is efficient and ubiquitous, but it may require some care to maintain its advantages throughout workflows that rely on multiple implementations. In this chapter, you've learned how to work around some common pitfalls and retain the benefits of a columnar format across language ecosystems.

Chapter 8

Scaling up and out across the data science workflow

In our blueprint application, we benefit from two kinds of parallelism: *scaling up*, or accelerating fine-grained tasks with specialized hardware or improved single-node performance, and *scaling out*, or dividing a problem into larger tasks and distributing these across machines. In our federation and analytics applications, Apache Spark itself provides scale-out parallelism and the RAPIDS Accelerator for Apache Spark provides scale-up parallelism for tasks within Spark. In our feature extraction and machine learning applications, we benefit from scaling up with RAPIDS.ai and scaling out with Dask. However, the benefits of each kind of parallelism are not uniform across the data science workflow. In this chapter, you'll learn how to most effectively use your compute resources across the application lifecycle and data science workflow.

8.1 Address limitations by scaling up and out

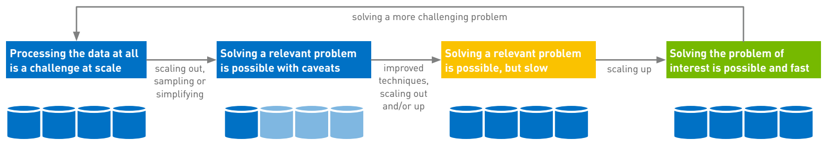
Consider some of the high-level reasons why computer system performance is limited:

- *finite storage* at a given level of the memory hierarchy limits how many values can be accessed efficiently,
- *finite execution capacity* at the instruction level, limiting how much computation can be done in a particular amount of time, and
- *finite bandwidth* between CPU or GPU and memory, between memory and storage, or between nodes, limiting how quickly data can be transferred from where it is to where it can be accessed efficiently.

Different workloads will be affected by different root causes of poor performance, and thus will benefit from different strategies to improve performance. Broadly, if a workload is storage- or memory-bound, it may benefit from scaling out, if a workload is execution-bound, it may benefit from scaling up, and if a workload is bandwidth-bound, either strategy may be appropriate depending on specifics of the workload.

Scaling up can take the form of hardware acceleration, which includes superscalar and multithreaded processors, SIMD and vector processing, and GPU acceleration. We can also construe scaling up more broadly to encompass factors that improve single-node performance without explicitly introducing parallelism, like larger and faster memories or disks (to improve overall throughput), faster system buses (to increase bandwidth across the memory hierarchy), optimized library code (to improve the efficiency of frequently-accessed routines), and efficient storage formats like Parquet (in order to improve I/O bandwidth and memory system performance). Scaling out involves coordinating work across multiple threads, GPUs, and machines and is typically most effective when a problem cannot be solved with the resources of a single node or when the coordination and communication costs involved in orchestrating coarse-grained tasks across multiple nodes are outweighed by the benefits of executing each task concurrently with independent resources. Scale-out strategies may also be useful for allowing a memory-hungry workload to execute to completion on a single node by dividing the problem into tasks that can execute independently and serially given the memory constraints of that node.

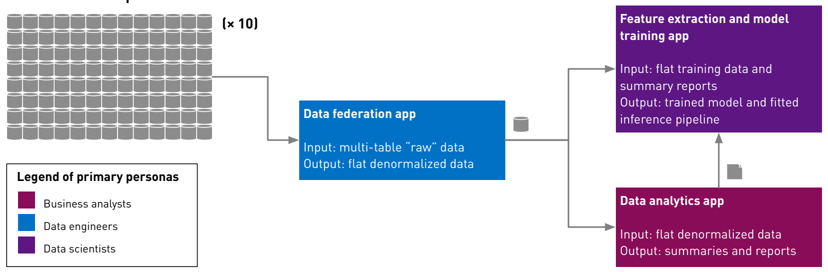
For many workloads, a combination of scale-up and scale-out strategies will be appropriate. Apache Spark itself is a scale-out framework that uses some scale-up techniques (including native code generation for optimized query processing). When combined with the RAPIDS Accelerator plugin, Spark is more clearly both a scale-out framework (via distributed processing) and a scale-up framework (via GPU acceleration of query operators).


solving a more challenging problem
Processing the data at all is a challenge at scale
scaling out, sampling or simplifying
Solving a relevant problem is possible with caveats
improved techniques, scaling out and/or up
Solving a relevant problem is possible, but slow
scaling up
Solving the problem of interest is possible and fast

Often, applications evolve along multiple axes: first, it is challenging to process the data at scale at all; then, scalability improvements make it possible to solve an interesting problem on a subset of the data (or a coarse approximation of the problem of interest on the whole dataset). With further advances in scalability, it may be possible to solve the problem of interest more precisely on the full dataset, albeit slowly; finally, further engineering or algorithmic improvements make it possible to solve the problem of interest on the full dataset quickly. Scale-out techniques, scale-up techniques, better algorithms, or some combination of the three may be responsible for each leap in an application's evolution. Of course, once it is possible to solve a given problem quickly at scale, our focus often shifts to solving a more challenging version of that problem: considering more variables, processing more data, making low-latency predictions, identifying a more precise approximation, and so on.

8.2 Know when to scale up and when to scale out

Because different parts of our workload are constrained by different aspects of our hardware and architecture, they benefit more or less from different kinds of parallelism. Some parts of our workload are constrained by memory availability (given working set sizes to execute queries) and some are more constrained by capacity to execute compute-intensive tasks quickly. In general, data federation and analytics workloads can benefit more from scale-out techniques and machine learning techniques can benefit from scale-up techniques, but there are circumstances in which both kinds of workloads can benefit from both kinds of parallelism.


(× 10)
Data federation app
Input: multi-table "raw" data
Output: flat denormalized data
Feature extraction and model training app
Input: flat training data and summary reports
Output: trained model and fitted inference pipeline
Data analytics app
Input: flat denormalized data
Output: summaries and reports
Legend of primary personas
Business analysts
Data engineers
Data scientists

The figure above represents the data volumes operated on by the federation, analytics, and machine learning applications in our blueprint. (Note that, in order to make the figure fit on the page, the federation input data volume is actually one-tenth the scale it should be – the analytics and ML apps operate on roughly one-one-thousandth the volume of data as the federation app.)

Scale analytics and federation out and up

Unsurprisingly, the federation application benefits from scaling out, whether with multiple threads or multiple machines. As we've seen, it also benefits from scaling up, by adding GPUs to each executor, but because the federation work is not particularly compute-intensive, the benefits from scaling up may be more modest.

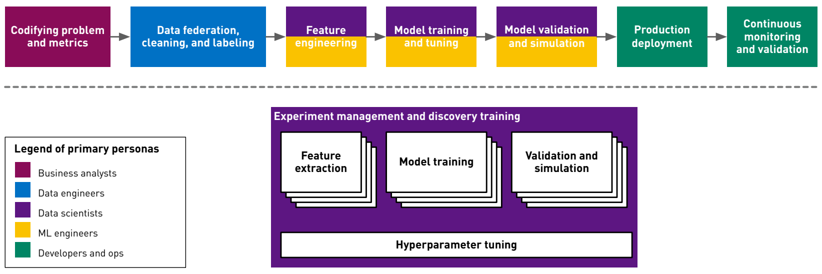
The analytic application deals with a tiny fraction – less than a thousandth – of the data that the federation application does, but its work is more computationally involved, since the data cube operation requires examining counts of churning and retained customers for each combination of values from select columns. On-line analytics jobs are often also well-suited for coarse-grained parallel execution.

When we perform feature extraction and encoding, we further distill the federated structured data into feature vectors. Therefore, the input data volume for model training proper is even smaller than the output from the federation job. In addition, model training can be far more computationally intensive than data federation or even analytics. Finally, model training often does not benefit as much from coarse-grained parallelism as it does from running many fine-grained compute kernels in parallel.

Individual model training tasks can effectively scale up

The amount of raw or structured data our system must handle for federation or analytics is typically far greater than the amount of data our system will train a machine learning model on. However, the demands on single-node compute performance are far greater once we're training a model. In the earlier, data engineering-focused parts of the workflow, we are limited by our data throughput; in later, data-science-focused parts of our workflow, we are limited by our compute operation throughput. Concretely, the federation and analytics application operates on roughly one hundred and forty gigabytes of structured data, but model training operates on under one hundred *megabytes* of feature vectors. While we are able to scale federation and analytics both out and up, seeing benefits both from running on an eight-node cluster and from adding GPUs to each node, training an individual model by itself operates on a relatively small volume of data and is best served by scaling up.

Experimental machine learning can scale out and up


Codifying problem and metrics
Data federation, cleaning, and labeling
Feature engineering
Model training and tuning
Model validation and simulation
Production deployment
Continuous monitoring and validation
Legend of primary personas
Business analysts
Data engineers
Data scientists
ML engineers
Developers and ops
Experiment management and discovery training
Feature extraction
Model training
Validation and simulation
Hyperparameter tuning

However, data scientists rarely train a single model in isolation – rather, they often train many models in parallel in order to identify the best-performing *hyperparameter settings*, or parameters supplied to the model training algorithm. Data scientists can also train multiple independent models together that will be federated as an ensemble. Each of these training tasks is an excellent candidate for scaling up, but the entire experimental workload, which consists of potentially many independent training runs, is an excellent candidate for scaling up.

Interactive data science exploration benefits from scaling up

In the process of identifying a successful model and feature extraction approach, a data scientist may train several auxiliary models that don't necessarily make it into a production system. Rather, these auxiliary models make it easier to understand the structure of the data or the behavior of the model. For example, fitting a UMAP or t-SNE model to a high-dimensional dataset may make it possible to visualize the essential structure of a high-dimensional dataset in two or three dimensions. Similarly, techniques like LIME or SHAP train auxiliary models to explain the behavior of other models. Because these techniques are on the critical path of a creative professional, we can dramatically improve practitioner velocity by making these techniques as fast as possible, typically by scaling them up.

8.3 Schedule heterogeneous workloads to take advantage of available compute resources

In this chapter, you've seen that the workloads supporting different stages of our discovery workflow will benefit more or less from different kinds of parallelism. Analytics and federation workloads that process a large amount of data will benefit both from scaling up with GPU acceleration, and out across multiple GPUs. Training an individual machine learning model will typically have a much smaller working set size; since the data involved will often fit in GPU memory on a single GPU, classical machine learning model training is often a better candidate for scaling up. (Some model training, including complex deep learning models or very large tree ensembles, may still benefit from scaling out even at modest data volumes.) Experiment orchestration, which can involve training many models to identify optimal hyperparameter settings, can scale both out and up.

You'll want to rely on the flexibility of a general-purpose scheduler like YARN or Kubernetes to effectively utilize a cluster. Plan to allocate more resources for federation and analytics jobs than for individual model training jobs. Interactive data science and exploratory analytics both benefit from scaling up, in order to reduce latency and improve practitioner experience, but present additional challenges to schedule clusters effectively, since interactive workloads can run indefinitely and necessarily exhibit lower utilization than batch workloads.

Chapter 9

Machine learning training and inference

In previous chapters, we've seen an overall architecture for churn modeling and prediction, how to accelerate data federation and analytics, how to integrate these components, and how to effectively use our compute resources by scaling up and out. In this chapter, we'll focus on the remaining parts of our workflow and system, highlighted below, and discuss how our application trains a model and performs inference.

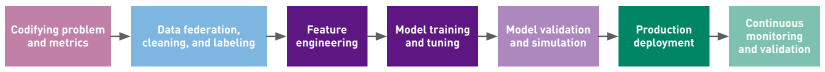
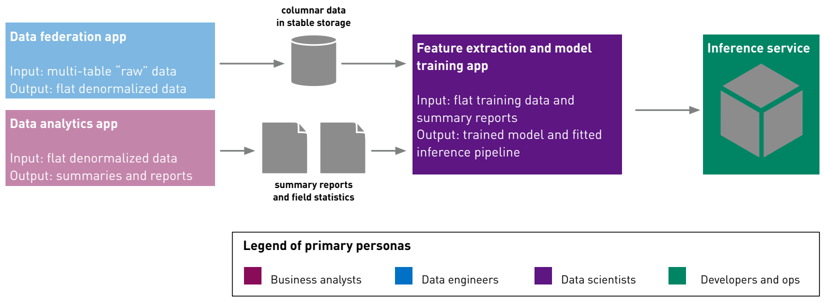

Codifying problem and metrics
Data federation, cleaning, and labeling
Feature engineering
Model training and tuning
Model validation and simulation
Production deployment
Continuous monitoring and validation

Figure 9.1: Workflow stages covered in this chapter

9.1 Communicate structured feature metadata from federation and analysis


Data federation app
Input: multi-table "raw" data
Output: flat denormalized data
Data analytics app
Input: flat denormalized data
Output: summaries and reports
columnar data in stable storage
summary reports and field statistics
Feature extraction and model training app
Input: flat training data and summary reports
Output: trained model and fitted inference pipeline
Inference service
Legend of primary personas
Business analysts
Data engineers
Data scientists
Developers and ops

Recall that our data federation application produces flat tabular data where each row contains all the data we have about a given customer, federated and aggregated from multiple source tables. Our analytics application produces both summary reports and a machine-readable metadata report. While the former may inform a data scientist's modeling activity (by suggesting features that are positively correlated with churn), the latter is intended to be consumed directly by modeling code.

Our metadata report contains the following information in a JSON object:

- the *schema* for the federated data, including column names and data types;
- *empirical cumulative distributions* for numeric columns, so that downstream modeling code can readily identify where a given value falls in the distribution of observed values (it's often more useful to know, for example, that a value is in the ninety-seventh percentile than that it's two standard deviations above the mean);
- the *percentage of true values* for boolean columns;
- a structure describing potential *encodings* for various columns, identifying columns that are likely categorical values, likely numerical values, or likely unique values; and
- a *count of distinct customers* in the output data.

In some organizations, the kinds of metadata we generate in our report may be collected and maintained separately from the data (in a metadata catalog) rather than passed directly to modeling code. In some cases, a metadata catalog will consist of prescriptive expectations as to the kinds and distributions of values rather than descriptive summaries of observed data. Both kinds of metadata reports can be useful, but the descriptive approach we take has the advantage of always describing the data we are actually operating upon.

Data scientists may collect similar information as part of their initial exploratory analysis and then use this information to inform downstream modeling. However, unless this process is formalized and automated, the information derived from an initial exploration may become stale with changes to upstream data or to the federation job itself, just as prescriptive expectations in a metadata catalog can become stale. Out-of-date information about schemas, encodings, or distributions is likely to be less useful than no information at all, and unexpected changes to data formats and distributions are an important source of silent bugs in machine learning systems.

By generating summary reports on newly-federated data with automated analysis code, we eliminate a source of potential bugs, ensure that downstream analyses are consuming fresh metadata, and remove a dependency on human discipline in our pipeline. We can also compare distributions over time as we collect new data and detect upstream data errors by identifying unexpected divergences.

9.2 Accelerate encoding categorical and boolean columns

Appropriately encoding numerical features (like a customer's monthly bill, lifetime account value, or length of tenure) is necessary to get acceptable performance out of many model-training algorithms. Depending on the model and the feature, a linear transformation, normalization, or recentering may be most appropriate. As we saw in Chapter 7, though, many of our features are *categorical features*, which do not take values that have a natural numeric interpretation.

Application programmers are comfortable turning arbitrary values from a finite domain into numbers by creating an enumerated type or by enumerating them, as the following hypothetical code snippet does:

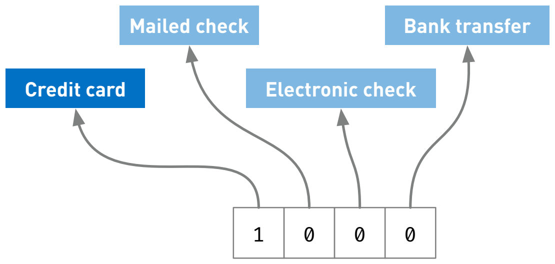
```
class PaymentType(object):
    CREDIT_CARD = 0
    MAILED_CHECK = 1
    BANK_TRANSFER = 2
    ELECTRONIC_CHECK = 3
```

With these constant definitions, a program can consistently represent distinct payment types as unique small integers. If this technique sounds familiar, it should – it's essentially a manual version of the dictionary encoding that Parquet applies to low-cardinality columns to improve space efficiency.

Most machine learning models ultimately expect numeric inputs, but dictionary encoding alone is, in general, insufficient to treat categorical features. This is the case since the numbers we assign to individual feature values are not meaningful. As an example, consider the informal encoding above: we've assigned values arbitrarily and a priori and thus there's no sense in which a bank transfer (2) is twice the magnitude of a mailed check (1), or in which an electronic check (3) is more similar to a bank transfer (2) than it is to a credit card (0). Without some additional domain-specific information in our encoding, the only meaningful comparison we can make between feature values is whether or not they are identical.

One-hot encoding is one way to turn our informal dictionary encoding into a meaningful way to represent categorical features. In this case, we'll be turning a feature with *k* possible values into a *k*-bit vector in which exactly one bit is set. We could also turn such a feature into a *k-1*-bit vector in which *at most* one bit is set and in which a vector with *no* bits set corresponded to one of the feature values. The approach we're taking is most appropriate because we're training an ensemble of decision trees and each branch in a decision tree can depend on only one comparison. So while it'd be fine for a branch to depend on, for example, whether or not bit 2 was set to 1, we couldn't have a single branch depending on if *all* bits were 0. (By contrast, the *k-1*-bit approach would be appopriate if we were going to train a linear model.)

The value we get from our initial encoding will be the index of the set bit, as in the figure below, which shows a vector that encodes a credit card payment type.



There are two ways to accelerate one-hot encoding with RAPIDS: you can use the `get_dummies` method in cuDF or the `OneHotEncoder` class in cuML. The following code excerpt shows how to use the former, assuming `path` is the input path of the data set, `binary_cols` is a list of column names that have binary values, and `ohe_cols` is a list of column names that have categorical values.

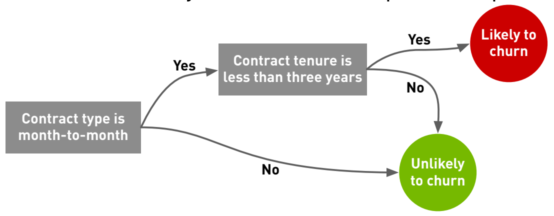
```
import cudf

# path is the input path of the dataset
# binary_cols is a list of column names that have binary values
# ohe_cols is a list of column names that have categorical values

data = cudf.read_parquet(path, read_dictionary=binary_cols+ohe_cols)
data_enc = cudf.get_dummies(data, columns=ohe_cols, dtype='bool')

# encode boolean or binary-valued columns as True/False
for cc in binary_cols:
    data_enc[cc] = data_enc[cc].cat.codes.astype('bool')
```

This code is very similar to code a data scientist might write in pandas. Recall that we know the types of columns because of the metadata produced by our analytics application. We discussed the need for `read_dictionary` in Chapter 7. Because boolean and two-valued columns in our source data are encoded as strings — including `Yes` / `No`, `0` / `1`, and `Male` / `Female`, for example — we also convert these to boolean values using the categorical codes.

One-hot encoding is not always the right approach for treating categorical values; we have used it here because most of the categoricals in this problem have relatively small domains (most frequently, three or four elements). If we use one-hot encoding to transform a categorical with many potential values, like postal codes, we will greatly increase the dimensionality and the sparsity of our data. Other approaches, like cardinal encoding (replacing values with their counts from the training set), quantizing values by clustering or exploiting semantic hierarchies, or target encoding (replacing values with the fraction of times they correlate with a true label) may be more effective for categoricals with large domains.

9.3 Scale tree ensemble training up with GPUs

If we were going to define a set of static rules to predict churning customers, we might imagine inspecting the data, identifying the features most strongly correlated with churning, and then constructing a set of questions about an individual customer so that the answers would identify whether or not that customer was likely to churn. A simple example is in the *decision tree* below:


Contract type is month-to-month
Yes
Contract tenure is less than three years
Yes
Likely to churn
No
No
Unlikely to churn

We can use machine learning to train a decision tree automatically by optimizing the set of questions we ask to minimize error. The main advantages of decision trees are that they are simple and interpretable; you can always explain why you arrived at a given conclusion by listing the questions you asked from the root of the tree to a decision. The main disadvantage of decision trees is that they are inflexible and thus prone to both imprecision and overfitting. In the former case, we are unable to train a model with sufficiently acceptable predictive performance on our training set. In the latter case, we are unable to train a model that has comparable predictive performance on novel data as it does on the training set; in other words, it has failed to generalize from the training examples.

By combining several decision trees in an *ensemble*, we can eliminate the disadvantages of simple individual decision trees. The basic technique involves training many decision trees to solve the same problem but in different ways — whether each focuses on a subset of input columns or on a subset of training examples — so that each individual tree has relatively weak performance. When it comes time to make predictions, the technique will take the consensus of the predictions from each individual decision tree.

The particular tree ensemble method we use in this application is *gradient-boosted trees*, as implemented by the XGBoost library. At a high level, XGBoost is able to optimize for one of many objective functions (not just accuracy) and can focus later rounds of training on examples that previous rounds performed poorly on. It is many data scientists' default choice for classification and regression problems on structured data.

A full discussion of XGBoost is outside the scope of this book, but the XGBoost documentation has an excellent introduction to XGBoost and this NVIDIA developer blog post by Rory Mitchell explains how XGBoost benefits from GPU acceleration. XGBoost itself supports accelerated inference as well as accelerated training. Complex models or use cases requiring high-throughput inference will benefit from the RAPIDS Forest Inference Library, which accelerates production inference of XGBoost models.

We can train a tree ensemble model using GPU-accelerated XGBoost. We'll use the native XGBoost interface through Dask even though we're operating on a single node for now, just to make it easier to scale up and out in the future:

```
from dask.distributed import Client
import xgboost
```

Our next step is to initialize a Dask client. As we discussed in Chapter 8, because of the scale of this problem, training on a single GPU is actually faster than training on a cluster of CPUs (or a cluster of GPUs).

```
# we're using only one worker because the scale of this problem
client = Client(n_workers=1, threads_per_worker=8)
```

From our training data (which we loaded and encoded in an earlier excerpt), we can assemble a training set. For training purposes, we create a `DaskDeviceQuantileDMatrix` from our data. This precomputes an efficient internal representation for numeric columns and can dramatically reduce memory consumption; however, we cannot use it for inference.

```
# - data_enc is the data frame we populated
#   in the previous section
# - feature_cols is a list of feature columns
# - label_col is the label column (i.e., "Churn")

train = data_enc.drop["customerID"].sample(.7)

X = train[feature_cols].astype('float32')
y = train[label_col].astype('float32')
dtrain = xgboost.dask.DaskDeviceQuantileDMatrix(client, X, y)
```

Finally, we're able to train the model using a GPU-accelerated algorithm:

```
xgb_model = \
    xgboost.dask.train(client,
                       tree_method='gpu_hist',
                       dtrain)['booster']
```

If you've used XGBoost in the past, you may have used it via its scikit-learn compatible interface. You'll want to use the native interface instead because it provides the most support for features necessary for scaling up and out.

9.4 Use Dask for multi-GPU and multi-node training

Modeling customer churn in the way we do in this blueprint does not benefit from multiple nodes or multiple GPUs, even at reasonable scale, since the entire dataset fits in GPU memory and the processing itself is not complex enough to saturate a GPU. However, many problems will benefit from multiple

GPUs or even from multiple nodes. For some problems, the volume of data arriving on a data scientist's desk may be so great that even exploratory work and feature encoding might benefit from scaling out. If we need to scale out to multiple GPUs or multiple nodes, we'll be able to use the same Dask interface we used above, with a few minor changes and additional considerations.

To run on multiple GPUs on a single node, we'll connect to a Dask `LocalCUDACluster`. The following example shows how to use a node with four GPUs:

```
from dask_cuda import LocalCUDACluster

cuda_cluster = LocalCUDACluster(n_workers=4, threads_per_worker=8)
client = Client(cuda_cluster)
```

To run with multiple nodes, we'll need a multinode Dask cluster. Since the details of how to do so are necessarily specific to your environment, the best way to learn how to get started with multinode Dask is to consult the Dask documentation, which covers scheduling jobs on on Kubernetes, on YARN, and on public cloud providers.

Chapter 10

Next steps

In this book, you've learned what to look out for as you build a contemporary accelerated machine learning system. You've learned about the cross-functional teams who build machine learning systems and about how their work comes together to solve business problems. You've seen how to accelerate data engineering and analytics workloads with the RAPIDS Accelerator for Apache Spark and how to accelerate machine learning training and inference with RAPIDS and XGBoost. We've seen how to ensure that the different components in our pipeline are interoperable and how to best manage compute resources for the different demands of data engineering and machine learning workloads.

We've intentionally focused on the connections, challenges, and curiosities in this process, but any practitioner knows that there's the potential to dive arbitrarily deep on any of the stages here. Now that you know what to watch for as you put these pieces together, why not use this framework to explore a really interesting new data engineering or modeling technique?

Accelerated computing makes it possible to solve problems we couldn't have tackled before. While we're able to train a model for customer churn with a single GPU faster than we could with a cluster of powerful CPUs, there are many applications of classical machine learning that are impractical without GPU acceleration. Other applications are complex enough to benefit from multiple GPUs or multiple nodes. (Many interesting deep learning techniques, of course, are only feasible at all with GPU acceleration.) Now that you know how to accelerate entire pipelines, why not tackle a problem that has seemed out of reach in the past?

Finally, many practitioners know the benefits of getting into a state of flow, when everything is focused and seems to be working perfectly. Interruptions and delays can kill flow, though. By making your exploratory, interactive work seamless and fast — whether you're an analyst waiting for SQL queries to return results or a data scientist waiting for a manifold projection to power visualization — GPU acceleration makes it possible to stay in that state of flow and try new ideas as quickly as you can think them up.

To learn more about some of the technologies we've discussed, visit the following sites:

- NVIDIA Developer
- RAPIDS
- RAPIDS Accelerator for Apache Spark
- Dask

Acknowledgments

The blueprint application described in this book was a team effort. Bao Tong developed the code and techniques corresponding to feature extraction, model training, and inference. Eric Rife helped harden some of our automation and scripting, developed documentation, and provided valuable feedback early in the development cycle.

I am grateful to Jake Schmitt, Bao Tong, Mike Beaumont, Chase Hooley, and Belen Tegegn for feedback on this book.