Tiago da Silva
João Alcindo Ribeiro de Azevedo
Germano Andrade Brandão

Relatório
Computação Escalável

Rio de Janeiro
2022

Sumário

1 Introdução

Trataremos aqui das decisões de projetos adotadas ao longo do trabalho, procurando esclarecer o nosso ponto de vista acerca de cada escolha. Além disso, toda a estrutura será detalhada, de forma a ser possível entender bem como funciona todo o processo de simulação de ponta a ponta. Por fim, mostraremos como o trabalho foi adaptado para que fosse processado em nuvem, utilizando majoritariamente os serviços da Amazon AWS.

2 Pipeline

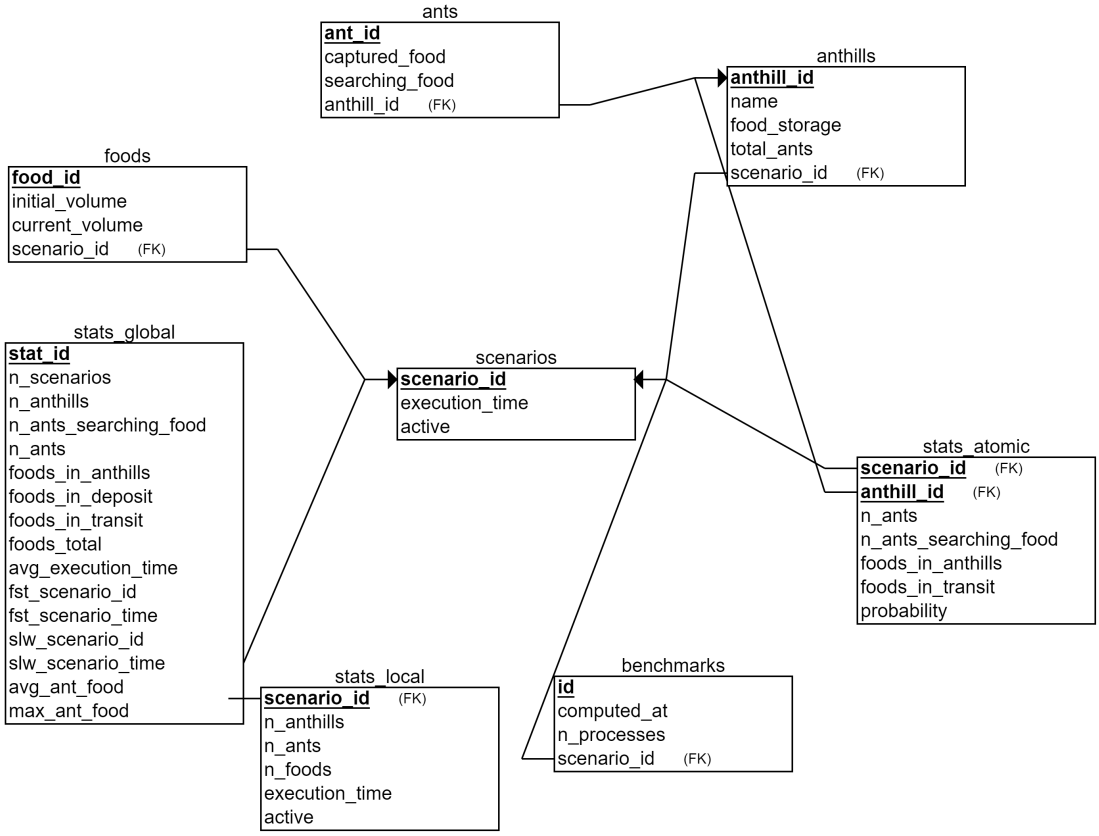


Figura 1: Modelo relacional do banco de dados utilizado para a captura persistente dos dados gerados pela simulação.

Escolhemos uma pipeline de extração, transformação e carregamento (ETL, em oposição a ELT); importantemente, esta escolha está amarrada à contemplação de que os procedimentos para serializar os dados e os tornar receptivos a um banco

de dados relacional gozam de intensidade computacinoal enxuta. Utilizamos, para isso, um banco de dados alicerçado no modelo relacional da Figura 1. Nas seções seguintes, desta maneira, descrevemos a implementação de cada estágio de nosso pipeline ETL.

Neste ínterim, advogamos o modelo escolhido e apresentamos o framework geral para controlar a comunicação entre os processos responsáveis pelo inserção dos dados em um banco de dados. Inicialmente, as tabelas `scenarios`, `ants`, `anthills` e `foods` almejam incorporar as informações brutas geradas pelas simulações; isso enseja a expansão da amplitude de análises subsequentes. As tabelas com prefixo `stats_`, por outro lado, são atualizadas periodicamente e permitem, desta maneira, o acesso, em diferentes aspectos de granularidades[1], às informações por usuários (na Seção 5, exploramos o desenho de uma interface que direciona o usuário ao exame dos cenários correntes e passados). Alternativamente, poderíamos

1. implementar uma base de dados analítica, com um modelo dimensional que combinasse parciomoniosamente as tabelas para o cômputo das quantidades solicitadas nos requisitos da modelagem;

2. e, mais disruptivamente, optar pela utilização de um banco de dados noSQL, com um pipeline ELT, em que os dados seriam armazenados e, em seguida, processados.

Nestas condições, nossas escolhas estão amarradas a compleições objetivas: com respeito a 1, precisaríamos comportar, em nosso sistema, um mecanismo de transferência de informações entre um banco de dados operacional e um analítico, trascendendo, em nossa opinião, o objetivo desta avaliação, que não inclui a modelagem dimensional; com respeito a 2, nossas breves exposições a bancos de dados noSQL (como o MongoDB) culminaram em nossas reticências em os introduzir neste sistema. Neste cenário, nossas escolhas por um banco de dados relacional estão alicerçadas em sua conveniência e na existência de múltiplas ferramentas que permitem a sua integração em um sistema distribuído.

Adicionalmente, a distribuição da pipeline ETL em múltiplos nós de computação implicou a escolha de um mecanismo de comunicação; em nossa opinião, este mecanimos deveria (1) ser escalável (isto é, as mensagens, se existirem, têm de ser persistentes e as chamadas, assíncronas; assim, garantimos que os clientes podem interagir com a interface e que seus dados serão inseridos na base de dados), (2) ser transparente (equivalentemente, a sua incorporação ao sistema existente

[1]Explicitamente, escrevemos `global` para as análises globais, que tangenciam todos os cenários executados; `local`, para as análises que consolidam os aspectos de cada cenário; e `atomic`, para as análises que conformam as instâncias que apontam para os formigueiros.

deveria exigir modificações enxutas no código-fonte) e (3), mais importante, compatível com um cenário em que a estrutura subjacente das mensagens (como o tipo de dado e o tamanho) é difusa e variável. Vislumbramos, nestas condições, um tripleto de procedimentos, que descrevemos em seguida.

1. *Remote Procedure Calls.* Apesar de as chamadas remotas permitirem transparência na comunicação e serem compatíveis com o nosso cenário, elas não são apropriadas em um cenário em que não podemos garantir que o receptor será executado quando o requerimento for enviado, como o nosso, em que múltiplos clientes podem interagir com uma quantidade enxuta de servidores.

2. *Message Passing Interface.* A interface de passagem de mensagens, por outro lado, não é compatível com o sistema que desejamos, na medida em que sua utilização está amarrada à estimativas do tamanho da mensagem, que, em nosso cenário, não estão disponíveis. Mais crucialmente, a introdução desse mecanismo exige, em alguns casos, a escolha cautelosa dos processos responsáveis pelo processamento das mensagens. Logo, em oposição à sua abstração conveniente e à sua flexibilidade, esta interface é inadequada para o que almejamos.

3. *Publish-Subscribe.* Nestas circunstâncias, o publish-subscribe permite, em oposição às chamadas de procedimento remotas, o processamento subsequente de mensagens, que são persistentes, em um momento posterior ao seu envio pelo produtor (comunicação assíncrona); em oposição à interface de passagem de mensagens, ele é consistente com o nosso sistema, com uma implementação que não subverte o código-fonte inicial.

Neste sentido, escolhemos o mecanismo de comunicação por mensagens persistentes publish-subscribe para o gerenciamento da comunicação entre os nós de processmaneto. Utilizamos, para isso, a biblioteca `Celery`, em `Python`, que providencia uma implementação eficiente deste mecanismo. Precisávamos, assim, de um broker para garantir a integração consistente de múltiplas máquninas em um sistema distribuído; escolhemos, portanto, a implementação do `RabbitMQ` por sua popularidade[2] e por sua acessibilidade em servições de computação em nuvem, como a AWS (AmazonMQ; veja a Seção **??**).

Nas seções seguintes, desta maneira, descreveremos os procedimentos utilizados para extração dos dados da simulação, sua subsequente transformação em um formato compatível com um banco de dados relacional, e seu carregamento em tabelas. Conforme enfatizamos na Seção **??**, utilizamos o serviço RDS da AWS

[2]Serviços populares, com uma comunidade mais extensa, permitem que usuários possam instanciar uma rede de apoio mútuo que incrementa a eficiência do desenvolvimento de *software*.

para a implementação do banco de dados, que é escalável e contempla as operações do PostgreSQL, com uma interface adequada em `Python`.

2.1 Extração

Logo, com o objetivo de descrever a extração dos dados gerados pelos clientes, encetamos com uma descrição de seu funcionamento. Explicitamente, implementamos uma simulação de um sistema emergente, em que formigas de distintos formigueiros interagem com um mapa finito para capturar alimentos e os conduzir a seus depósitos. Cada simulação, neste caso, é caracterizada por um conjunto de informações, como a quantidade de formigas e a extensão da execução, que precisamos extrair e direcionar a um consumidor que as alocará em uma base de dados. A etapa de extração, deste modo, consistiu na aplicação de procedimentos que garantissem a consistência entre as implementações dos mecanismos de comunicação utilizados, o `Celery`, e os processos de geração de dados de nossa implementação. Especificamente, serializamos os atributos das classes subjacentes à simulação e utilizamos o mecanismo de comunicação para transferir os dados do JSON culminante para o servidor, em que, conforme descrevemos na seção seguinte, eles são transformados em um formato cartesiano.

2.2 Transformação

Nestas veredas, a etapa de transformação está amarrada à captura das mensagens empilhadas no broker e seu subsequente processamento, com a modificação de um JSON não estruturado para enformar instâncias compatíveis com o modelo relacional da Figura 1. Dicotomicamente, este procedimento é computacionalmente enxuto, na medida em que ele é equivalente à remodelação do formato de uma JSON, e, em contraste, ele é volumoso: em cada iteração da simulação, delineada como um momento em que todas as formigas executaram suas ações, os clientes se comunicam com o broker, e o intervalo entre mensagens sequentes está na ordem de milissegundos. Importantemente, esta verificação estende nossas motivações para a escolha de um mecanismo de comunicação assíncrono e orientado a mensagens persistentes, como o publish-subscribe.

2.3 Carregamento

Enfaticamente, portanto, os dados transformados estavam receptivos ao seu carregamento na base de dados; executamos, para isso, as operações de inserção e de atualização propiciadas pelo sistema de gerenciamento de banco de dados utilizado, PostgreSQL, e sua interface em `Python`, `psycopg2` (versão 2.9.3).

Crucialmente, a eficiência deste estágio está correlacionada aos protolocos de comunicação utilizados pela rede subjacente; em nossos experimentos, contudo, este aspecto não consistia em um gargalo computacional.

3 Processamento em Paralelo

Em conjunto com uma pipeline ETL, implementamos procedimentos para processar os dados, computando as 14 quantidades solicitadas nos requerimentos. Com o objetivo de garantir escalabilidade e transparência, escolhemos o `Spark`, e `PySpark`, para executar esta tarefa. Em maior detalhe, o `PySpark` atua independentemente do `Celery`, capturando periodicamente os dados disponíveis na base de dados e executando algoritmos para computar as quantidades sumárias; inconvenientemente, os estágios de IO, em que os dados são lidos e, em seguida ao processamento, escritos na base de dados, concentram o maior consumo temporal[3].

No entanto, em oposição a uma pipeline ETL, escolhemos não introduzir o `PySpark` em um servidor na nuvem da AWS operando continuamente (veja a seção seguinte). Com efeito, objetivamente, existem restrições pecuniárias no consumo do serviços da AWS que controlam a disponibilidade do que podemos inserir nesta plataforma[4]. Mais subjetivamente, o `PySpark` foi desenhado para o processamento de volumes expandidos de dados; contrastivamente, nossas tabelas contemplam milhares de instâncias.

4 Armazenamento em nuvem (AWS)

O processamento e o armazenamento locais podem ser vantajosos por ter uma velocidade alta e não depender de conexões externas. No entanto, essas vantagens podem acabar rapidamente ao passo que a quantidade a ser processada/armazenada começa a aumentar. Nesse sentido, uma das alternativas mais adotadas atualmente é a de computação em nuvem, onde um ambiente vasto com soluções prontas ou customizáveis são disponibilizadas e de forma escalável, de acordo com a demanda.

Dito isso, para esse trabalho utilizamos alguns dos serviços de nuvem da Amazon Web Services, a fim de agregar mais robustez ao nosso pipeline de dados.

[3]Neste cenário, não introduzimos procedimentos *ad hoc* para o processamento de dados, e não objetivamos controlar a redundância na tabela. Manifestamente, a modelagem dimensional para o usufruto analítico dos dados poderia ser substancialmente mais adequada; contudo, ela transcende nossos objetivos.

[4]Por experimentação, nós instanciamos um cluster no AWS EMR; contudo, nós o encerramos para garantir a persistência do serviço para outros usuários.

Assim, veremos agora quais recursos foram utilizados e informações das instâncias criadas em cada um deles.

4.1 AmazonMQ

Uma vez que utilizamos o RabbitMQ como broker, temos o AmazonMQ como serviço de gerenciamento de mensagens na AWS para criar uma instância com o RabbitMQ.

Ao instanciar com as opções devidamente escolhidas, ficamos com as seguintes informações (Tabela 1).

Broker engine	Engine version	Instance type
`RabbitMQ`	`3.9.16`	`mq.t3.micro`

Tabela 1: Informações da instância na AmazonMQ

A partir disso, podemos nos conectar com o broker a partir dos dados a seguir (Tabela 2).

ARN (Amazon Resource Name)	RabbitMQ web console	port
`arn:aws:mq:us-east-1:676432491375:broker:TJG:b-7182fca9-4c07-4bfa-be01-72310cb18d60`	`https://b-7182fca9-4c07-4bfa-be01-72310cb18d60.mq.us-east-1.amazonaws.com`	`5671`

Tabela 2: Informações para conexão

Para acessar o console, basta utilizar as credenciais a seguir (Tabela 3).

Username	Password
`username`	`passwordpassword`

Tabela 3: Credenciais RabbitMQ

4.2 RDS

O Amazon RDS (*Relational Database Service*) é uma coleção de serviços gerenciados que facilita a configuração, operação e escalabilidade de bancos de dados na nuvem. A partir dele, é possível operar dentre diversos mecanismos, como MySQL, PostgreSQL, MariaDB, SQL Server e etc.

No nosso caso, para utilização do banco de dados relacional, uma instância de PostgreSQL foi iniciada na Amazon RDS. Desse modo, falaremos aqui de suas especificações e deixaremos as credenciais de acesso.

Assim, a instância foi iniciada com as informações abaixo (Tabela 4):

Engine	Engine version	Type	Allocated storage
PostgreSQL	13.4	db.t3.micro	20 GiB

Tabela 4: Informações da instância na RDS

Feito isso, podemos acessá-la através das seguintes credenciais (Tabela 5):

Credencial	Valor
DB instance identifier	database-postgres-tjg
Username	postgres
Password	passwordpassword
Host (endpoint)	database-postgres-tjg.cvb1csfwepbn.us-east-1.rds.amazonaws.com
Port	5432

Tabela 5: Credenciais de acesso à instância da AWS RDS

A partir daí, utilizamos o Python 3, através da biblioteca psycopg2 para se conectar com a instância e realizar todas as operações de manipulação e inserção/-consulta dos dados no banco.

```python
# importing
import psycopg2
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT

# creating a connection
db = psycopg2.connect(
        host = "database-postgres-tjg.cvb1csfwepbn.us-east-1.rds.amazonaws.com",
        user="postgres",
        password="passwordpassword",
        database="operational_tjg" #database previously created
        )

# setting the isolation level
db.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)

# initializing cursor
cursor = db.cursor()
```

Figura 2: Conexão com a instância através do Python utilizando a biblioteca psycopg2

4.3 ECR

Para hospedar imagens Docker, utilizamos o Amazon ECR (*Elastic Container Registry*) que é um registro de contêiner totalmente gerenciado da Amazon AWS.

Nesse sentido, criamos dois repositórios, um para o *celery* e outro para o *spark*.

Repository name	URI
`tjg-celery`	`676432491375.dkr.ecr.us-east-1.amazonaws.com/tjg-celery`
`tjg-spark`	`676432491375.dkr.ecr.us-east-1.amazonaws.com/tjg-spark`

Tabela 6: Informações das imagens

4.4 ECS

Para gerenciar os containers, utilizamos o Amazon ECS (*Elastic Container Service*), onde podemos executar tarefas dentro do cluster.

Cluster	Cluster ARN	Launch type
`TJG`	`arn:aws:ecs:us-east-1:676432491375:cluster/TJG`	`FARGATE`[5]

Tabela 7: Informações do cluster

Com isso, ficamos aptos a definir as *tasks*.

Definimos duas *tasks* no cluster, a primeira com respeito ao worker do celery e a segunda com relação ao Spark.

Task definition name	OS family	Compatibilities	Task memory (MiB)	Task CPU (unit)
`TJG-Worker-task`	`Linux`	`EC2, FARGATE`	`1024`	`512`
`TJG-Spark-task-2`	`Linux`	`EC2, FARGATE`	`16384`	`4096`

Tabela 8: Informações das *tasks*

Para executar e manter um número específico de instâncias de uma *task* simultaneamente no cluster, utilizamos o ECS service. Confguramos dois services para realizar as *tasks* comentadas acima. Abaixo temos as especificações de cada *service*.

O `TJG-Worker-Service` performou como planejado, executando suas tarefas e realizando o seu trabalho no pipline. Já no `TJG-Spark-Service`, tivemos algumas

[5] O AWS Fargate é uma tecnologia que pode ser usada com o Amazon ECS para executar contêineres sem a necessidade de gerenciar servidores ou clusters de instâncias do Amazon EC2.

Service name	Task definition	Service type	Launch type
TJG-Worker-Service	TJG-Worker-task	REPLICA	FARGATE
TJG-Spark-Service	TJG-Spark-task-2	REPLICA	FARGATE

Tabela 9: Informações das tasks

inconsistências, os status das tasks ficavam oscilando em PENDING, RUNNING e PROVISIONING. Tentamos, aumentar a memória e CPU dessas tasks, porém continuamos observando as inconsistências.

5 Interface com o usuário

Além disso, desenhamos um par de interfaces para o usuário interagir com o nosso sistema: em uma delas, uma aplicação web, propiciamos visualizações de cada cenário e de quantidades sumárias, globais (de todos os clientes) e locais (de cada cliente, corrente ou passado); em outra, ensejamos que a consulta à base de dados por comandos da linguagem SQL. A aplicação gráfica está disponível na Figura [figura], e é descrita doravante. Em um console, contudo, os procedimentos são mais elementares; por exemplo, se quisermos consultar a quantidade de formigas contempladas por todos os cenários, executamos

```
python query.py --query 'SELECT COUNT(*) FROM ants;'
```

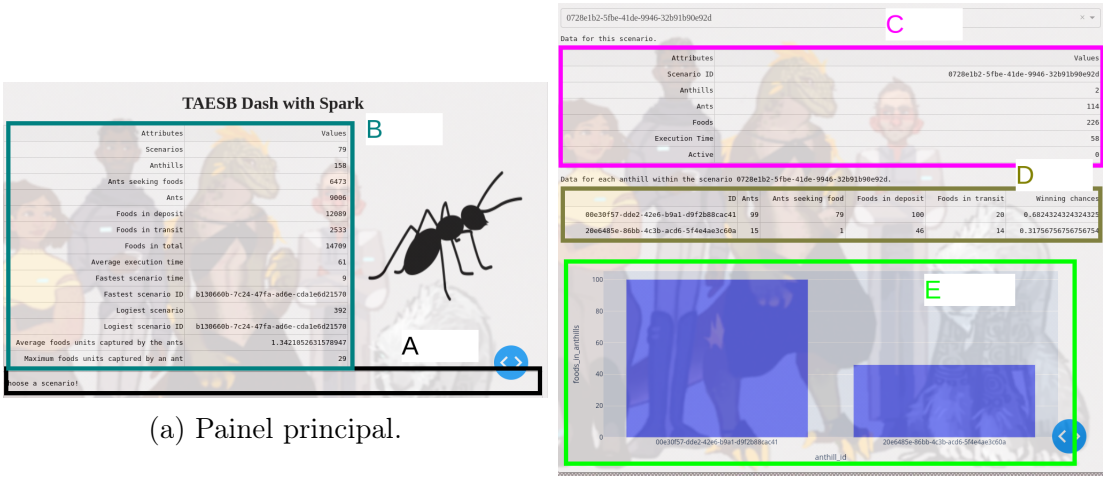
e, se almejarmos identificar a lista de cenários executados, e seus tempos de execução (medidos em quantidades de iterações), utilizamos

```
python query.py --query 'SELECT * FROM scenarios;'
```

A interação gráfica, contudo, é mais direcionada aos objetivos elicitados a priori da modelagem do sistema. Na Figura [figura], factualmente, expomos as quantidades globais, que consolidam todos os cenários, em uma tabela; também pertimos, alternativamente, a escolha de um cenário e o acompanhamento de suas informações específicas e de seus formigueiros (inserimos, para isso, um gráfico de barras para enfatizar a discrepância entre os depósitos de comidas). Com o objetivo de a iniciar, execute, no diretório raiz da aplicação,

```
python spark_dash.py
```

e direcione seu navegador no sítio impresso na tela. Com ênfase, perecba que existe um par de dissonâncias entre os cenários atuais e as tabelas: por um lado, o processamento independente pelo `Spark` suscita uma discrepância temporal entre as informações na base de dados e as informações factuais; por outro, a interface gráfica também goza de uma taxa de atualização que expande a distinção entre a realidade e os registros.



(a) Painel principal.

(b) Dados locais.

Figura 3: Interface gráfica que propicia a interação do usuário com os dados gerados pela simulação: em (a), desenhamos o painel principal e, em (b), as informações locais do cenário solicitado pelo usuário. Em mais detalhes, o usuário escolhe, em `A`, o cenário que ele almeja escrutinar; em `B`, ele é exposto às condições globais. Nestas condições, a escolha de um cenário culmina em um painel (`C`) em que podemos vislumbrar informações a respeito dele, como a quantidade de formigas que participam de sua simulação; em `D`, ampliamos a granularidade de nossa análise e inserimos os dados amarrados aos formigueiros incluídos no cenário escolhido; em `E`, desenhamos uma figura para enfatizar a discrepância entre os depósitos de alimentos de cada formigueiro, alicerce do cômputo de suas chances de vitória.

Instâncias	Tempo de execução
1	25.22
3	15.86
5	14.98
7	14.44

Tabela 10: Benchmarks para o pipeline ETL distribuído; perceba que incrementar a quantidade de instâncias não é, objetivamente, uma estratégia estritamente dominante, conforme a métrica utilizada.

A Benchmarks

Nesta seção, mensuramos o tempo de execução da pipeline ETL para distintas quantidades de instâncias em execução no cluster da AWS. Encetamos, neste sentido, descrevendo os procedimentos para o cômputo desta métrica: cada cenário envia, em algum momento, uma mensagem inicial para inserir suas instâncias (formgiueiros, comidas e formigas) nas base de dados, garantindo a consistência das inserções a atualizações seguintes; nós registramos este momento. As mensagens subsequentes, assim, também estão equipadas de um atributo temporal, que é inserido na tabela `benchmarks`. Portanto, caracterizamos o *tempo de execução de um cenário* como a diferença entre o momento em que sua mensagem mais recente foi processada e o momento em que sua mensagem inicial foi processada; o *tempo médio de processamento*, desta maneira, é igual à média dos tempos de execução de todas os cenários. Na Tabela 10, inserimos, para quantidades incrementais de instâncias participando do pipeline, estas medidas; executamos, para isso, 40 cenários.

Crucialmente, o incremento da quantidade de instâncias no cluster não consiste de uma escolha uniformemente adequada; a priori, com efeito, poderíamos acreditar que, para uma quantidade arbitrariamente grande de nós de computação, poderíamos gozar de um tempo de processamento arbitrariamente enxuto. No entanto, as adições marginais de eficiência computacional são *decrescentes*; possivelmente, essa verificação está amarrada à existência de custos fixos de comunicação e e de transferência de informações e, também, ao controle de acesso concorrente pelo sistema de gerenciamento de banco de dados que são incólumes, em alguma extensão, à quantidade de instâncias participando da pipeline. Em nosso cenário, particularmente, o volume de dados é enxuto; o custo variável, amarrado à transformação das informações, é, acreditamos, eclipsado pelos mecanismos subjacentes ao sistema distribuído; isso é, aliás, bastante enfático na implementação do Spark. Mais geralmente, o desenho de sistemas distribuídos exige uma escolha cautelosa entre taxa de processamento e custo; clusters tremendamente expansivos não conforma, em geral, a escolha mais apropriada.