O teorema CAP afirma que um banco de dados não pode garantir consistência, disponibilidade e tolerância a partições ao mesmo tempo. Mas você não pode sacrificar a tolerância à partição (veja aqui e aqui), então você deve fazer uma troca entre disponibilidade e consistência. Gerenciar essa compensação é o foco central do movimento NoSQL.

Consistência significa que depois de fazer uma gravação bem-sucedida, as leituras futuras sempre levarão essa gravação em consideração. Disponibilidade significa que você sempre pode ler e gravar no sistema. Durante uma partição, você só pode ter uma dessas propriedades.

Os sistemas que optam pela consistência em vez da disponibilidade têm de lidar com alguns problemas complicados. O que você faz quando o banco de dados não está disponível? Você pode tentar armazenar em buffer as gravações para mais tarde, mas corre o risco de perdê-las se perder a máquina com o buffer. Além disso, o buffer de gravações pode ser uma forma de inconsistência porque um cliente pensa que uma gravação foi bem-sucedida, mas ainda não está no banco de dados. Como alternativa, você pode retornar erros ao cliente quando o banco de dados estiver indisponível. Mas se você já usou um produto que lhe dizia para "tentar novamente mais tarde", sabe como isso pode ser agravante.

A outra opção é escolher a disponibilidade em vez da consistência. A melhor garantia de consistência que estes sistemas podem fornecer é a "consistência eventual". Se você usar um banco de dados eventualmente consistente, às vezes você lerá um resultado diferente do que acabou de escrever. Às vezes, vários leitores lendo a mesma chave ao mesmo tempo obterão resultados diferentes. As atualizações podem não ser propagadas para todas as réplicas de um valor, então você acaba com algumas réplicas recebendo algumas atualizações e outras réplicas recebendo atualizações diferentes. Cabe a você reparar o valor assim que detectar que os valores divergiram. Isso requer rastrear o histórico usando relógios vetoriais e mesclar as atualizações (chamado "reparo de leitura").

Acredito que manter a consistência eventual na camada de aplicação é um fardo muito pesado para os desenvolvedores. O código de leitura e reparo é extremamente suscetível a erros do desenvolvedor; se e quando você cometer um erro, reparos de leitura defeituosos introduzirão corrupção irreversível no banco de dados.

Portanto, sacrificar a disponibilidade é problemático e a consistência eventual é muito complexa para construir aplicativos de maneira razoável. No entanto, essas são as duas únicas opções, então parece que estou dizendo que você está condenado se fizer isso e condenado se não fizer isso. O teorema CAP é um facto da natureza, então que alternativa poderá existir?

Existe outra maneira. Você não pode evitar o teorema CAP, mas pode isolar sua complexidade e evitar que ele sabote sua capacidade de raciocinar sobre seus sistemas. A complexidade causada pelo teorema CAP é um sintoma de problemas fundamentais na forma como abordamos a construção de sistemas de dados. Dois problemas se destacam em particular: o uso de estados mutáveis em bancos de dados e o uso de algoritmos incrementais para atualizar esse estado. É a interação entre esses problemas e o teorema CAP que causa complexidade.

Neste post vou mostrar o projeto de um sistema que supera o teorema CAP evitando a complexidade que ele normalmente causa. Mas não vou parar por aí. O teorema CAP é um resultado sobre o grau em que os sistemas de dados podem ser tolerantes a falhas de máquinas. No entanto, existe uma forma de tolerância a falhas que é muito mais importante do que a tolerância a falhas da máquina: a tolerância a falhas humanas. Se há alguma certeza no desenvolvimento de software, é que os desenvolvedores não são perfeitos e os bugs inevitavelmente chegarão à produção. Nossos sistemas de dados devem ser resilientes a programas com erros que gravam dados incorretos, e o sistema que vou mostrar é tão tolerante a falhas humanas quanto possível.

Esta postagem desafiará suas suposições básicas sobre como os sistemas de dados devem ser construídos. Mas ao quebrar as nossas formas atuais de pensar e reimaginar como os sistemas de dados devem ser construídos, o que emerge é uma arquitetura mais elegante, escalável e robusta do que alguma vez imaginou ser possível.

O que é um sistema de dados?

Antes de falarmos sobre design de sistema, vamos primeiro definir o problema que estamos tentando resolver. Qual é o propósito de um sistema de dados? O que são dados? Não podemos sequer começar a abordar o teorema CAP a menos que possamos responder a estas questões com uma definição que encapsule claramente cada aplicação de dados.

As aplicações de dados variam desde armazenamento e recuperação de objetos, junções, agregações, processamento de fluxo, computação contínua, aprendizado de máquina e assim por diante. Não está claro se existe uma definição tão simples de sistemas de dados – parece que a gama de coisas que fazemos com os dados é muito diversa para ser capturada com uma única definição.

No entanto, existe uma definição tão simples. É isso:

```
Query = Function(All Data)
```

É isso. Esta equação resume todo o campo de bancos de dados e sistemas de dados. Tudo na área – os últimos 50 anos de RDBMS, indexação, OLAP, OLTP, MapReduce, ETL, sistemas de arquivos distribuídos, processadores de fluxo, NoSQL, etc. – é resumido por essa equação de uma forma ou de outra.

Um sistema de dados responde a perguntas sobre um conjunto de dados. Essas perguntas são chamadas de "consultas". E esta equação afirma que uma consulta é apenas uma função de todos os dados que você possui.

Esta equação pode parecer demasiado geral para ser útil. Não parece capturar nenhuma das complexidades do design do sistema de dados. Mas o que importa é que todo sistema de dados se enquadra nessa equação. A equação é um ponto de partida a partir do qual podemos explorar sistemas de dados, e a equação acabará por levar a um método para superar o teorema CAP.

Existem dois conceitos nesta equação: “dados” e “consultas”. Esses são conceitos distintos que muitas vezes são confundidos no campo de banco de dados, portanto, sejamos rigorosos sobre o que esses conceitos significam.

Dados

Vamos começar com “dados”. Um dado é uma unidade indivisível que você considera verdadeira por nenhuma outra razão além de sua existência. É como um axioma em matemática.

Existem duas propriedades cruciais a serem observadas sobre os dados. Primeiro, os dados são inerentemente baseados no tempo. Um dado é um fato que você sabe ser verdadeiro em algum momento. Por exemplo, suponha que Sally insira em seu perfil na rede social que ela mora em Chicago. Os dados que você obtém dessa entrada são que ela morava em Chicago no momento específico em que inseriu essas informações em seu perfil. Suponha que, posteriormente, Sally atualize a localização de seu perfil para Atlanta. Então você sabe que ela morava em Atlanta naquela época específica. O fato de ela morar em Atlanta agora não muda o fato de ela morar em Chicago. Ambos os dados são verdadeiros.

A segunda propriedade dos dados decorre imediatamente da primeira: os dados são inerentemente imutáveis. Devido à sua conexão com um determinado momento, a veracidade de um dado nunca muda. Não se pode voltar no tempo para alterar a veracidade de um dado. Isso significa que existem apenas duas operações principais que você pode realizar com os dados: ler os dados existentes e adicionar mais dados. CRUD se tornou CR.

Deixei de fora a operação "Atualizar". Isso ocorre porque as atualizações não fazem sentido com dados imutáveis. Por exemplo, "atualizar" a localização de Sally realmente significa que você está adicionando um novo dado informando que ela mora em um novo local em um momento mais recente.

Também deixei de fora a operação "Excluir". Novamente, a maioria dos casos de exclusões são melhor representados como a criação de novos dados. Por exemplo, se Bob parar de seguir Mary no Twitter, isso não muda o fato de que ele a seguia. Então, em vez de excluir os dados que dizem que ele a segue, você adicionaria um novo registro de dados que diz que ele deixou de segui-la em algum momento.

Existem alguns casos em que você deseja excluir dados permanentemente, como regulamentos que exigem que você limpe os dados após um determinado período de tempo. Esses casos são facilmente suportados pelo projeto do sistema de dados que vou mostrar, portanto, para fins de simplicidade, podemos ignorá-los.

Essa definição de dados é quase certamente diferente daquela com a qual você está acostumado, especialmente se você vem do mundo dos bancos de dados relacionais, onde as atualizações são a norma. Há duas razões para isso. Primeiro, esta definição de dados é extremamente genérica: é difícil pensar num tipo de dados que não se enquadre nesta definição. Em segundo lugar, a imutabilidade dos dados é a propriedade chave que iremos explorar na concepção de um sistema de dados tolerante a falhas humanas que supere o teorema CAP.

Consulta

O segundo conceito da equação é a “consulta”. Uma consulta é uma derivação de um conjunto de dados. Nesse sentido, uma consulta é como um teorema em matemática. Por exemplo, "Qual é a localização atual de Sally?" é uma

consulta. Você calcularia essa consulta retornando o registro de dados mais recente sobre a localização de Sally. As consultas são funções do conjunto de dados completo, portanto podem fazer qualquer coisa: agregações, unir diferentes tipos de dados e assim por diante. Portanto, você pode consultar o número de usuárias do seu serviço ou consultar um conjunto de dados de tweets sobre quais tópicos têm sido tendências nas últimas horas.

Defini uma consulta como uma função no conjunto de dados completo. É claro que muitas consultas não precisam do conjunto de dados completo para serem executadas – elas só precisam de um subconjunto do conjunto de dados. Mas o que importa é que a minha definição encapsula todas as consultas possíveis, e se quisermos vencer o teorema CAP, devemos ser capazes de fazê-lo para qualquer consulta.

Superando o teorema CAP

A maneira mais simples de calcular uma consulta é literalmente executar uma função no conjunto de dados completo. Se você pudesse fazer isso dentro de suas restrições de latência, estaria pronto. Não haveria mais nada para construir.

Obviamente, é inviável esperar que uma função em um conjunto de dados completo termine rapidamente. Muitas consultas, como aquelas que atendem um site, exigem tempos de resposta de milissegundos. No entanto, vamos fingir por um momento que você pode calcular essas funções rapidamente e ver como um sistema como esse interage com o teorema CAP. Como você verá, um sistema como esse não apenas supera o teorema CAP, mas o aniquila.

O teorema CAP ainda se aplica, então você precisa escolher entre consistência e disponibilidade. A beleza é que, uma vez que você decida a troca que deseja fazer, estará pronto. A complexidade que o teorema CAP normalmente causa é evitada pelo uso de dados imutáveis e consultas computacionais do zero.

Se você escolher consistência em vez de disponibilidade, não haverá muitas mudanças em relação a antes. Às vezes, você não conseguirá ler ou gravar dados porque negociou a disponibilidade. Mas para os casos em que a consistência rígida é uma necessidade, é uma opção.

As coisas ficam muito mais interessantes quando você escolhe disponibilidade em vez de consistência. Neste caso, o sistema é eventualmente consistente sem nenhuma das complexidades da consistência eventual. Como o sistema é altamente disponível, você sempre pode gravar novos dados e calcular consultas. Em cenários de falha, as consultas retornarão resultados que não incorporam dados gravados anteriormente. Eventualmente, esses dados serão consistentes e as consultas irão incorporar esses dados em seus cálculos.

A chave é que os dados são imutáveis. Dados imutáveis significam que não existe atualização, portanto é impossível que diferentes réplicas de um dado se tornem inconsistentes. Isso significa que não há valores divergentes, relógios vetoriais ou reparo de leitura. Da perspectiva das consultas, um dado existe ou não existe. Existem apenas dados e funções nesses dados. Não há nada que você precise fazer para impor consistência eventual, e a consistência eventual não atrapalha o raciocínio sobre o sistema.

O que causou complexidade antes foi a interação entre atualizações incrementais e o teorema CAP. As atualizações incrementais e o teorema CAP realmente não funcionam bem juntos; valores mutáveis requerem reparo de leitura em um sistema eventualmente consistente. Ao rejeitar atualizações incrementais, adotar dados imutáveis e calcular consultas do zero sempre, você evita essa complexidade. O teorema CAP foi derrotado.

Claro, o que acabamos de passar foi um experimento mental. Embora gostaríamos de poder calcular consultas do zero todas as vezes, isso é inviável. No entanto, aprendemos algumas propriedades importantes de como será uma solução real:

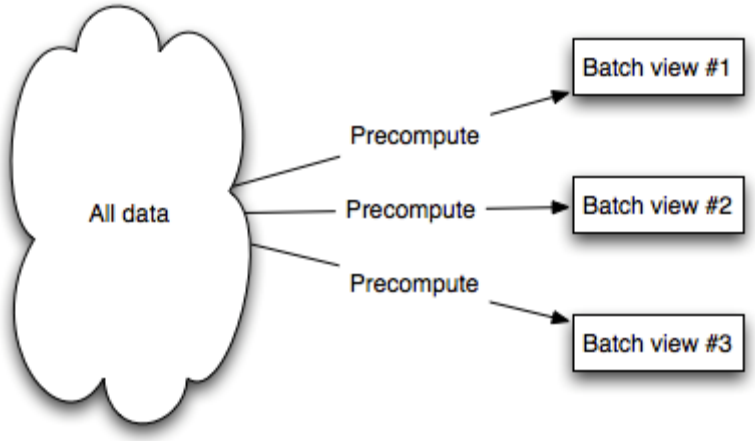
1. O sistema facilita o armazenamento e a escala de um conjunto de dados imutável e em constante crescimento
2. A principal operação de gravação é adicionar novos fatos imutáveis de dados
3. O sistema evita a complexidade do teorema CAP recomputando consultas a partir de dados brutos
4. O sistema usa algoritmos incrementais para reduzir a latência das consultas a um nível aceitável

Vamos começar nossa exploração da aparência de tal sistema. Observe que tudo daqui em diante é otimização. Bancos de dados, indexação, ETL, computação em lote, processamento de fluxo – todas essas são técnicas para otimizar funções de consulta e reduzir a latência a um nível aceitável. Esta é uma constatação simples, mas profunda. Os bancos de dados geralmente são considerados a peça central do gerenciamento de dados, mas na verdade eles são parte de um quadro maior.

Computação em lote

Descobrir como fazer com que uma função arbitrária em um conjunto de dados arbitrário seja executada rapidamente é um problema assustador. Então vamos relaxar um pouco o problema. Vamos fingir que não há problema em as consultas ficarem desatualizadas por algumas horas. Relaxar o problema dessa forma leva a uma solução simples, elegante e de uso geral para a construção de sistemas de dados. Depois, estenderemos a solução para que o problema não seja mais relaxado.

Como uma consulta é uma função de todos os dados, a maneira mais fácil de fazer com que as consultas sejam executadas rapidamente é pré-calculá-las. Sempre que há novos dados, você simplesmente recalcula tudo. Isso é viável porque relaxamos o problema para permitir que as consultas fiquem desatualizadas por algumas horas. Aqui está uma ilustração desse fluxo de trabalho:



Precomputation workflow

Para construir isso, você precisa de um sistema que:

1. Pode armazenar facilmente um conjunto de dados grande e em constante crescimento
2. Pode calcular funções nesse conjunto de dados de maneira escalonável

Esse sistema existe. É maduro, testado em batalhas em centenas de organizações e possui um grande ecossistema de ferramentas. Chama-se Hadoop . O Hadoop não é perfeito , mas é a melhor ferramenta disponível para processamento em lote.

Muitas pessoas dirão que o Hadoop só é bom para dados "não estruturados". Isto é completamente falso. O Hadoop é fantástico para dados estruturados. Usando ferramentas como Thrift ou Protocol Buffers , você pode armazenar seus dados usando esquemas ricos e evoluíveis.

O Hadoop é composto por duas partes: um sistema de arquivos distribuído (HDFS) e uma estrutura de processamento em lote (MapReduce). O HDFS é bom para armazenar uma grande quantidade de dados em arquivos de forma escalonável. MapReduce é bom para executar cálculos nesses dados de maneira escalonável. Esses sistemas atendem perfeitamente às nossas necessidades.

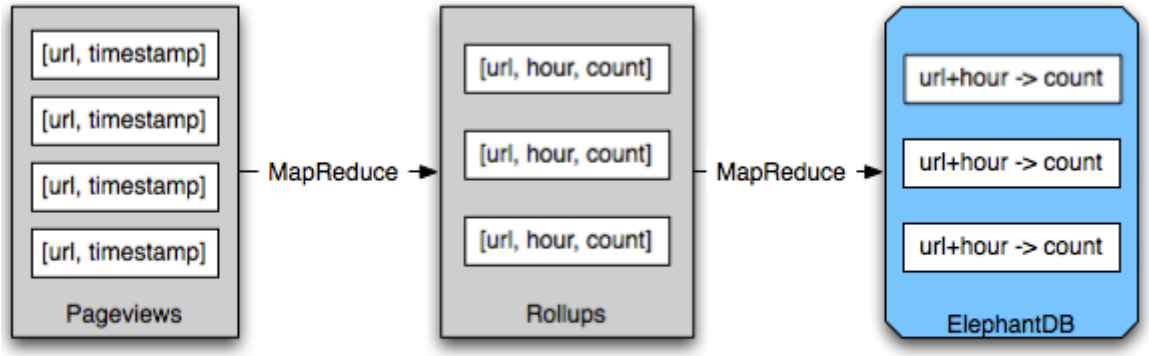
Armazenaremos dados em arquivos simples no HDFS. Um arquivo conterá uma sequência de registros de dados. Para adicionar novos dados, basta anexar um novo arquivo contendo novos registros de dados à pasta que contém todos os dados. Armazenar dados como esses no HDFS resolve o requisito de "Armazenar um conjunto de dados grande e em constante crescimento".

Pré-computar consultas a partir desses dados é igualmente simples. MapReduce é um paradigma expressivo o suficiente para que quase qualquer função possa ser implementada como uma série de tarefas MapReduce. Ferramentas como Cascalog , Cascading e Pig facilitam muito a implementação dessas funções.

Finalmente, você precisa indexar os resultados da pré-computação para que os resultados possam ser acessados rapidamente por um aplicativo. Existe uma classe de bancos de dados que são extremamente bons nisso. ElephantDB e Voldemort somente leitura são especializados na

exportação de dados de chave/valor do Hadoop para consultas rápidas. Esses bancos de dados oferecem suporte a gravações em lote e leituras aleatórias e *não* oferecem suporte a gravações aleatórias. As gravações aleatórias causam a maior parte da complexidade nos bancos de dados; portanto, por não suportarem gravações aleatórias, esses bancos de dados são extraordinariamente simples. O ElephantDB, por exemplo, tem apenas alguns milhares de linhas de código. Essa simplicidade faz com que esses bancos de dados sejam extremamente robustos.

Vejamos um exemplo de como o sistema em lote se encaixa. Suponha que você esteja criando um aplicativo de análise da web que rastreia visualizações de páginas e queira poder consultar o número de visualizações de páginas durante qualquer período de tempo, com granularidade de uma hora.



Batch workflow example

Implementar isso é fácil. Cada registro de dados contém uma única visualização de página. Esses registros de dados são armazenados em arquivos no HDFS. Uma função que acumula visualizações de páginas por URL por hora é implementada como uma série de trabalhos MapReduce. A função emite pares chave/valor, onde cada chave é um `[URL, hour]` par e cada valor é uma contagem do número de visualizações de página. Esses pares chave/valor são exportados para um banco de dados ElephantDB para que um aplicativo possa obter rapidamente o valor de qualquer `[URL, hour]` par. Quando um aplicativo deseja saber o número de visualizações de páginas em um intervalo de tempo, ele consulta o ElephantDB para obter o número de visualizações de páginas em cada hora nesse intervalo de tempo e os soma para obter o resultado final.

O processamento em lote pode calcular funções arbitrárias em dados arbitrários, com a desvantagem de que as consultas ficam desatualizadas por algumas horas. A "arbitrariedade" de tal sistema significa que ele pode ser aplicado a qualquer problema. Mais importante ainda, é simples, fácil de entender e totalmente escalonável. Basta pensar em termos de dados e funções e o Hadoop cuida da paralelização.

O sistema em lote, CAP e tolerância a falhas humanas

Até agora tudo bem. Então, como o sistema em lote que descrevi se alinha com o CAP e atende ao nosso objetivo de ser tolerante a falhas humanas?

Vamos começar com o PAC. O sistema em lote acaba sendo consistente da maneira mais extrema possível: as gravações sempre levam algumas horas para serem incorporadas às consultas. Mas é uma forma de consistência

eventual que é fácil de raciocinar porque você só precisa pensar nos dados e nas funções desses dados. Não há reparo de leitura, simultaneidade ou outros problemas complexos a serem considerados.

A seguir, vamos dar uma olhada na tolerância humana a falhas do sistema em lote. A tolerância a falhas humanas do sistema em lote é a melhor possível. Existem apenas dois erros que um ser humano pode cometer em um sistema como este: implantar uma implementação com erros de uma consulta ou gravar dados incorretos.

Se você implantar uma implementação com bugs de uma consulta, tudo o que você precisa fazer para consertar as coisas é consertar o bug, implantar a versão corrigida e recalcular tudo do conjunto de dados mestre. Isso funciona porque as consultas são funções puras.

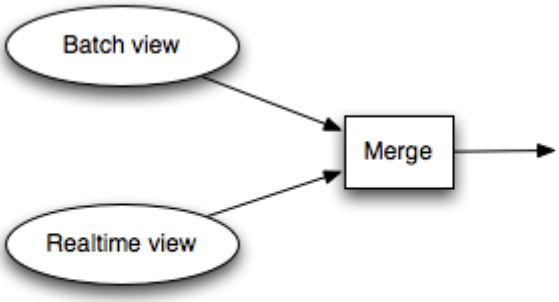
Da mesma forma, gravar dados inválidos tem um caminho claro para a recuperação: exclua os dados inválidos e pré-calcule as consultas novamente. Como os dados são imutáveis e o conjunto de dados mestre é apenas anexado, a gravação de dados incorretos não substitui ou destrói dados válidos. Isso contrasta fortemente com quase todos os bancos de dados tradicionais, onde se você atualizar uma chave, perderá o valor antigo.

Observe que o versionamento de linhas do tipo MVCC e HBase não chega perto desse nível de tolerância a falhas humanas. O versionamento de linhas MVCC e HBase não mantém os dados para sempre: depois que o banco de dados compacta a linha, o valor antigo desaparece. Somente um conjunto de dados imutável garante que você tenha um caminho para a recuperação quando dados incorretos forem gravados.

Camada em tempo real

Acredite ou não, a solução em lote resolve quase completamente o problema de calcular funções arbitrárias em dados arbitrários em tempo real. Quaisquer dados com mais de algumas horas já foram incorporados às visualizações em lote, portanto, tudo o que resta a fazer é compensar as últimas horas de dados. Descobrir como fazer consultas em tempo real com base em algumas horas de dados é muito mais fácil do que fazê-lo no conjunto de dados completo. Esta é uma visão crítica.

Para compensar essas poucas horas de dados, você precisa de um sistema em tempo real que funcione em paralelo com o sistema em lote. O sistema em tempo real pré-calcula cada função de consulta para as últimas horas de dados. Para resolver uma função de consulta, você consulta a visualização em lote e a visualização em tempo real e mescla os resultados para obter a resposta final.

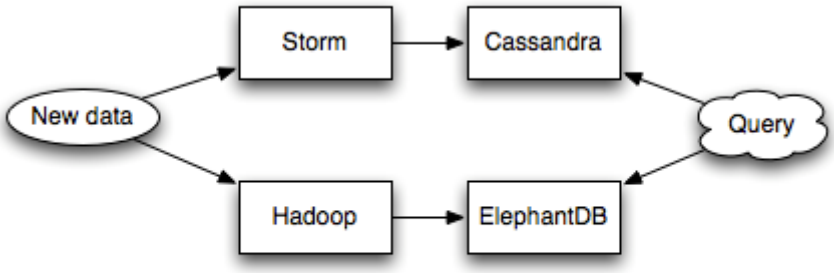


Computing a query

A camada em tempo real é onde você usa bancos de dados de leitura/gravação como Riak ou Cassandra, e a camada em tempo real depende de algoritmos incrementais para atualizar o estado nesses bancos de dados.

O análogo do Hadoop para computação em tempo real é o Storm . Eu escrevi o Storm para facilitar o processamento de grandes quantidades de dados em tempo real de uma forma escalonável e robusta. Storm executa cálculos infinitos em fluxos de dados e oferece fortes garantias no processamento dos dados.

Vamos ver um exemplo da camada em tempo real voltando ao exemplo em execução de consulta do número de visualizações de página de um URL em um intervalo de tempo.



Example batch/realtime architecture

O sistema em lote é o mesmo de antes: um fluxo de trabalho em lote baseado em Hadoop e ElephantDB pré-calcula a consulta para tudo, exceto as últimas horas de dados. Tudo o que resta é construir o sistema em tempo real que compense as últimas horas de dados.

Agregaremos as estatísticas das últimas horas no Cassandra e usaremos o Storm para processar o fluxo de visualizações de página e paralelizar as atualizações no banco de dados. Cada visualização de página leva a um contador para que uma `[URL, hour]`chave seja incrementada no Cassandra. Isso é tudo - Storm torna esse tipo de coisa muito simples.

Camada em lote + camada em tempo real, teorema CAP e tolerância a falhas humanas

De certa forma, parece que voltamos ao ponto de partida. Conseguir consultas em tempo real exigiu o uso de bancos de dados NoSQL e algoritmos incrementais. Isso significa que estamos de volta ao mundo complexo de valores divergentes, relógios vetoriais e reparo de leitura.

Porém, há uma diferença fundamental. Como a camada em tempo real compensa apenas as últimas horas de dados, tudo o que a camada em tempo real calcula é eventualmente substituído pela camada em lote. Portanto, se você cometer um erro ou algo der errado na camada em tempo real, a camada em lote irá corrigi-lo. Toda essa complexidade é transitória.

Isso não significa que você não deva se preocupar com o reparo de leitura ou com a eventual consistência na camada em tempo real. Você ainda deseja que a camada em tempo real seja o mais consistente possível. Mas quando você comete um erro, você não corrompe permanentemente seus dados. Isso remove um enorme fardo de complexidade de seus ombros.

Na camada de lote, você só precisa pensar nos dados e nas funções desses dados. A camada de lote é realmente simples de raciocinar. Na camada de tempo real, por outro lado, é necessário usar algoritmos incrementais e bancos de dados NoSQL extremamente complexos. Isolar toda essa complexidade na camada em tempo real faz uma enorme diferença na criação de sistemas robustos e confiáveis.

Além disso, a camada em tempo real não afeta a tolerância a falhas humanas do sistema. O conjunto de dados imutável somente anexado na camada de lote ainda é o núcleo do sistema, portanto, qualquer erro pode ser recuperado como antes.

Deixe-me compartilhar uma história pessoal sobre os grandes benefícios de isolar a complexidade na camada em tempo real. Eu tinha um sistema muito parecido com o que descrevi aqui: Hadoop e ElephantDB para a camada em lote e Storm e Cassandra para a camada em tempo real. Devido ao mau monitoramento de minha parte, acordei um dia e descobri que Cassandra estava sem espaço e estava expirando o tempo de cada solicitação. Isso fez com que minha topologia Storm falhasse e o fluxo de dados fizesse backup nas filas. As mesmas mensagens continuavam sendo repetidas (e falhando) continuamente.

Se eu não tivesse uma camada em lote, teria sido forçado a dimensionar e recuperar o Cassandra. Isso não é trivial. Pior ainda, grande parte do banco de dados provavelmente era imprecisa devido às mesmas mensagens serem repetidas muitas vezes.

Felizmente, toda essa complexidade foi isolada na minha camada de tempo real. Descarreguei as filas de backup na camada de lote e criei um novo cluster Cassandra. A camada em lote funcionou como um relógio e em poucas horas tudo voltou ao normal. Nenhum dado foi perdido e não houve imprecisão em nossas consultas.

Coleta de lixo

Tudo o que descrevi nesta postagem é construído sobre a base de um conjunto de dados imutável e em constante crescimento. Então, o que fazer se o seu conjunto de dados for tão grande que seja impraticável armazenar todos os

dados o tempo todo, mesmo com armazenamento escalável horizontalmente? Este caso de uso quebra tudo o que descrevi? Você deveria voltar a usar bancos de dados mutáveis?

Não. É fácil estender o modelo básico com "coleta de lixo" para lidar com esse caso de uso. A coleta de lixo é simplesmente uma função que coleta o conjunto de dados mestre e retorna uma versão filtrada do conjunto de dados mestre. A coleta de lixo elimina dados de baixo valor. Você pode usar qualquer estratégia que desejar para coleta de lixo. Você pode simular a mutabilidade mantendo apenas o último valor de uma entidade ou pode manter um histórico para cada entidade. Por exemplo, se você estiver lidando com dados de localização, talvez queira manter um local por pessoa por ano junto com o local atual. A mutabilidade é, na verdade, apenas uma forma inflexível de coleta de lixo (que também interage mal com o teorema CAP).

A coleta de lixo é implementada como uma tarefa de processamento em lote. É algo que você executa ocasionalmente, talvez uma vez por mês. Como a coleta de lixo é executada como uma tarefa de processamento em lote offline, ela não afeta a forma como o sistema interage com o teorema CAP.

Conclusão

O que dificulta os sistemas de dados escaláveis não é o teorema CAP. É a dependência de algoritmos incrementais e de estado mutável que leva à complexidade de nossos sistemas. Só recentemente, com o surgimento dos bancos de dados distribuídos, essa complexidade ficou fora de controle. Mas essa complexidade sempre existiu.

Eu disse no início deste post que desafiaria suas suposições básicas sobre como os sistemas de dados deveriam ser construídos. Transformei CRUD em CR, dividi a persistência em sistemas separados em lote e em tempo real e fiquei obcecado com a importância da tolerância humana a falhas. Foi necessária muita experiência suada ao longo dos anos para quebrar minhas antigas suposições e chegar a essas conclusões.

A arquitetura em lote/tempo real tem muitos recursos interessantes que ainda não abordei. Vale a pena resumir alguns deles agora:

1. **Flexibilidade algorítmica:** Alguns algoritmos são difíceis de calcular de forma incremental. Calcular contagens únicas, por exemplo, pode ser um desafio se os conjuntos de únicos ficarem grandes. A divisão lote/tempo real oferece flexibilidade para usar o algoritmo exato na camada de lote e um algoritmo aproximado na camada de tempo real. A camada em lote substitui constantemente a camada em tempo real, de modo que a aproximação é corrigida e seu sistema exibe a propriedade de "precisão eventual".
2. **As migrações de esquema são fáceis:** já se foram os dias das migrações de esquema difíceis. Como a computação em lote está no centro do sistema, é fácil executar funções no conjunto de dados completo. Isso facilita a alteração do esquema de seus dados ou visualizações.
3. **Análise ad hoc fácil:** a arbitrariedade da camada em lote significa que você pode executar qualquer consulta que desejar em seus dados. Como

todos os dados estão acessíveis em um único local, isso é fácil e conveniente.
4. **Autoauditoria:** ao tratar os dados como imutáveis, você obtém um conjunto de dados de autoauditoria. O conjunto de dados registra sua própria história. Já discuti como isso é importante para a tolerância a falhas humanas, mas também é muito útil para fazer análises.

Não afirmo ter "resolvido" o espaço do Big Data, mas estabeleci a estrutura para pensar sobre o Big Data. A arquitetura em lote/tempo real é altamente geral e pode ser aplicada a qualquer sistema de dados. Em vez de lhe dar um peixe ou uma vara de pescar, mostrei como fazer uma vara de pescar para qualquer tipo de peixe e qualquer tipo de água.

Há muito mais trabalho a ser feito para melhorar nossa capacidade coletiva de atacar problemas de Big Data. Aqui estão algumas áreas principais de melhoria:

1. **Modelos de dados expandidos para bancos de dados graváveis em lote e de leitura aleatória:** nem todos os aplicativos são suportados por um modelo de dados de chave/valor. É por isso que minha equipe está investindo na expansão do ElephantDB para oferecer suporte a pesquisas, bancos de dados de documentos, consultas de intervalo e muito mais.
2. **Melhores primitivas de processamento em lote** : o Hadoop não é o fim de tudo da computação em lote. Pode ser ineficiente para certos tipos de cálculos. Spark é um projeto importante que faz um trabalho interessante na expansão do paradigma MapReduce.
3. **Bancos de dados NoSQL de leitura/gravação aprimorados** : há espaço para mais bancos de dados com diferentes modelos de dados, e esses projetos em geral se beneficiarão de mais maturação.
4. **Abstrações de alto nível:** Uma das áreas mais interessantes de trabalho futuro são as abstrações de alto nível que mapeiam um componente de processamento em lote e um componente de processamento em tempo real. Não há razão para que você não deva ter a concisão de uma linguagem declarativa com a robustez da arquitetura em lote/tempo real.

Muitas pessoas desejam um banco de dados relacional escalonável. O que espero que você tenha percebido neste post é que você não quer isso de jeito nenhum! O big data e o movimento NoSQL pareciam tornar o gerenciamento de dados mais complexo do que era com o RDBMS, mas isso ocorre apenas porque estávamos tentando tratar o "Big Data" da mesma forma que tratamos os dados com um RDBMS: combinando dados e visualizações e confiando em algoritmos incrementais. A escala do big data permite construir sistemas de uma maneira completamente diferente. Ao armazenar dados como um conjunto de fatos imutáveis em constante expansão e ao incorporar a recomputação no núcleo, um sistema de Big Data é, na verdade, mais fácil de raciocinar do que um sistema relacional. E isso aumenta.