Rafael

Slide 1:

- Bom dia! A nossa apresentação vai ser sobre consistência transacional para computação Serverless, em que iremos abordar quais são as vantagens e as dificuldades deste sistema, bem como apresentar um sistema que resolve estas dificuldades, que é a HydroCache.

Slide 2:

- Nos últimos anos a computação Serverless, tem ganhado bastante atenção, mais em particular os sistemas FaaS, estes sistemas permitem que os programadores consigam fazer upload de funções arbitrárias e executá-las na Cloud sem a necessidade de fornecer ou manter servidores.
- Estas plataformas permitem que aplicações sejam construídas como uma composição de múltiplas funções.
- As plataformas FaaS são automaticamente escaláveis pois desagregam as camadas de execução e de armazenamento
- Estas funções são executadas de forma independente e podem ser corridas em diferentes máquinas, de forma a garantir um bom balanceamento de carga e tolerância a faltas.
- Um problema que existe nestas plataformas, é a latência de operações I/O.
- Uma solução que resolve este problema é o uso de caches, mas traz o problema de inconsistências entre elas.
- A solução proposta no artigo científico para resolver estes problemas é a HydroCache.

Slide 3:

- A HydroCache é implementada num sistema com arquitetura Cloudburst, em que a camada de armazenamento é implementada usando a Anna, que é uma Key-Value Store automaticamente escalável e a camada de execução é implementado usando um Cloudburst.
- O Cloudburst é constituído por Planeadores de Funções e por nós, em que cada nó contém Threads, que são os executores de funções e uma HydroCache.
- Todas os pedidos são recebidos por um planeador de funções que encaminha os pedidos para uma Thread.
- O sistema permite o encadeamento de funções, permitindo que os utilizadores registem composições de funções o chamado DAG de funções.
- Neste sistema a execução de um DAG é otimizada, sendo possível passar os resultados automaticamente de um executor de função para o próximo.

Slide 4:

- Consistência Causal existe se as leituras e as escritas respeitarem a relação de Lamport *"Happens-Before"*, que em português significa acontece antes.
- Ou seja, se a leitura de uma chave *ai* (em que *i* denota a versão da chave *a*) influencia uma escrita de uma chave *bj*, então *ai* acontece antes de *bj*, e, portanto, *bj* depende de *ai*.
- Se *ai* não depende de *aj*, nem *aj* depende de *ai*, então são concorrentes (*ai* ~ *aj*).

No nosso sistema uma chave *ki* tem 4 componentes:

- *k* que é identificador da chave.
- *VCk* que é um conjunto de relógios lógicos, com pares $< id, relógio >$, em que o *id* é o identificador da thread e o *relogio* representa o valor logico da Thread.
- *deps* que é um conjunto de dependências, com pares $< dep\_chave, VC >$ que representam as versões das chaves que o *ki* depende.
- *payload* que é o Valor associado à chave.

Slide 5:

- Consistências Causal+ é uma extensão da consistência causal, em que para alem de garantir causalidade garante que replicas, com a mesma chave vão eventualmente convergir para o mesmo valor.
- Para garantir isto, é registado uma política de resolução de conflitos na Anna que faz Merge de Versões Concorrentes de uma chave, armazenando os Payloads de ambas as versões, sendo possível às aplicações especificar qual o payload que querem.
- No caso de duas versões não serem concorrentes a mais atualizada é sobreposta à menos atualizada.

Slide 6:

- Consistência Causal Transacional é uma extensão da consistência causal+ que garante consistência em leituras e escritas para um conjunto de chaves.
- Mais especificamente, recebendo um conjunto de chaves de leitura, R, o TCC requer que R forme um Snapshot Causal.
- R só é um Snapshot Causal se e só se para qualquer par de chaves, *ai* e *bj* que pertencem a *R*, se *ak* for uma dependência de *bj* então ai não pode acontecer antes de *ak*. Por que se não, estaríamos a ler uma versão menos atualizada.

Slide 7:

- Agora vamos entrar em maior detalhe na HydroCache, mais em concreto em como é que é garantido TCC para funções individuais.
- Para garantir Snapshots Causais nas leituras, cada cache mantem um corte causal, *C* que requer que para qualquer dependência, *dj*, de qualquer chave em *C*, existe um *dk* que pertence a *C*, que sobre-excede *dj*, em que *dk* sobre-excede *dj* se e só se *dk* for igual a *dj* ou se *dj* acontece antes de *dk*.
- Existem 2 diferenças entre um corte e um Snapshot Causal:
- A primeira é que se uma chave estiver no corte então as suas dependências também estão.
- A segunda é que num corte não existe concorrência

Slide 8:

Como atualizar um corte local:

- A HydroCache inicialmente não contem dados, então cria trivialmente um corte *C*.
- Quando uma função pede uma chave b que não está presente no corte, a Cache vai buscar a versão *bj* à ana.
- Antes de devolver às funções, a Cache verifica se todas as dependências de *bj* sobre-excedem as chaves que já estão no corte.
- Se uma dependência, não for sobre-excedida, a cache vai buscar versões mais atualizadas à Anna, até isto acontecer.
- Isto é feito recursivamente até todas as dependências serem sobre-excedidas.
- No fim, a Cache atualiza *C* com as novas chaves.

Slide 9:

- A HydroCache garante Snapshots causais pois usa cortes e garante visibilidade atómica, ao fazer com que quando uma função escreve duas chaves, ambas ficam mutualmente dependentes uma da outra.
- Por forma a garantir que as escritas entre funções num DAG são mutualmente dependentes, todos as escritas são feitas no fim do DAG.
- Para alem disso, a HydroCache tem ainda um Garbage Collector, que remove dependências desnecessárias.
- A HydroCache garante ainda Tolerância a faltas.

Slide 10:

- Até agora abordamos TCC em funções individuais num só nó e vamos agora explicar como é garantido TCC em múltiplas funções que podem ser executadas em múltiplos nós, ou seja num DAG de funções.
- Para garantir TCC num DAG, uma hipótese seria fazer com que todas as caches se coordenassem e mantivessem um corte distribuído. Esta solução não é viável devido à quantidade de trafico que iria ser gerado entre milhares de nós.
- Neste sentido introduzimos protocolos MTCC que tentam resolver este problema enquanto ainda tentam minimizar a coordenação e sobrecarga do envio de dados entre caches.
- A ideia principal é que em vez de ser construído um corte distribuído, as caches colaboram para criar um Snapshot de cada DAG durante a execução, que leva a 2 grandes melhorias:
- A primeira é que os Snapshots são construídos para cada DAG e assim a comunicação para formar estes Snapshots é combinada com a execução normal de um DAG sem qualquer coordenação global
- A segunda é que os Snapshots apenas precisão de conter as chaves lidas pelo DAG em vez de todas.
- Iremos começar por abordar o protocolo centralizado, em que todas as funções num DAG são executadas por um só nó.
- Depois iremos introduzir 3 protocolos, que permitem que funções sejam executadas em diferentes nós.

Ricardo

Slide 11:

- O Centralizado ou, a sua abreviatura, CT, como referimos executa todas as funções de um dado DAG num só nó.
- Desta forma a complexidade do problema é bastante simplificada, uma vez que toda a execução do DAG recorre apenas a uma única cache. Não precisando assim de nos preocupar, com a dificuldade que requer a criação de um Snapshot Distribuído.
- Antes da execução do DAG é criado um Snapshot a partir do Corte Local que o dado nó possui, contendo o Readset de todas as funções que constituem o DAG.
- Este Snapshot serve todas as leituras executadas pelo DAG, garantindo assim que múltiplas leituras da mesma chave leiam o mesmo valor.
- Sendo um DAG totalmente executado num só nó, a escalabilidade do sistema é limitada, uma vez que o paralelismo de um DAG fica restringido ao número de Threads presentes num único nó.

Slide 12:

- Até este momento, vimos como criar um Snapshot num só nó.
- Vamos agora mostrar-vos alguns conceitos que serão necessários para percebermos o que se segue.
- Um Keyset é um conjunto de chaves sem versões específicas associadas.
- Um Versionset é o que associa um conjunto de chaves às suas respetivas versões.
- Um Keyset-Overlapping Cut é similar a um Corte, só que associado apenas às chaves que um dado Keyset contém, sendo as únicas dependências que este possui, as que igualmente constarem nesse mesmo Keyset. Ou seja, pode ser representado pela interceção de um Corte que um dado nó possui, com um Keyset. Esta interceção é realizada devido ao facto do propósito de um Keyset-Overlapping Cut ser o de criar um Snapshot, que por sua vez não necessita de outras chaves que não aquelas que serão utilizadas.
- É também importante referir, que um Keyset-Overlapping Cut constitui um Snapshot. E que graças às propriedades de um Keyset-Overlapping Cut a sua união com outros, continua a representar também ela um Keyset-Overlapping Cut. Esta propriedade será bastante útil para a criação de um Snapshot Distribuído.

Slide 13:

- O Otimista ou, a sua abreviatura, OPT, é o primeiro protocolo MTCC que vamos abordar.
- A ideia base do protocolo é começar automaticamente a executar as funções de um DAG, ao mesmo tempo que se verifica se ocorrem violações na construção do Snapshot Distribuído. Este Snapshot Distribuído, não é nada mais que a união de Keyset-Overlapping Cuts que cada nó realiza, associados às chaves que necessita para executar as funções do DAG.
- Este protocolo é particularmente vantajoso num sistema no qual as escritas são infrequentes, fazendo com que, Caches de diferentes nós, estejam em sincronia entre si, descartando deste modo a necessidade da construção de um Snapshot Distribuído em avanço.
- Contudo, em sistemas com escritas frequentes, Caches em diferentes nós, poderão não se encontrar em sincronia entre si, neste caso, essa dessincronia pode levar à violação do Snapshot Distribuído. Em algumas situações essa violação pode ser consertada, mas em certos casos, nos quais o seu ajuste já não é possível, é necessário recomeçar o DAG.
- Veremos agora, em que circunstancias é realizada a validação do Snapshot Distribuído.

Slide 14:

- Existem dois tipos de validações que podem ser aplicados ao Snapshot Distribuído. Fluxo Linear, que é aplicado quando a função a ser executada foi chamada pela sua predecessora no DAG. E o de Fluxo Paralelo, que é aplicado quando várias funções paralelas acumulam os seus resultados.
- Seja *Ri* uma Versionset associado ao Keyset das chaves que uma dada função *Fi* utiliza durante a sua execução.
- A validação do Fluxo Linear consiste em garantir que, dadas duas funções consecutivas no DAG, *Fu* e *Fd*, tal que *Fu* chama *Fd*, a união do Versionset *Ru* com o Versionset *Rd* que *Fd* gera, deve formar um Snapshot.
- Por outro lado, a validação do Fluxo Paralelo consiste em garantir que, dado um conjunto de funções *Fu* cujo resultado é acumulado, a união entre todos os seus Versionsets *Ru* forma também ela, um Snapshot.
- Se isso não se verificar, a execução do DAG é abortada, pois não é possível a formação de um Snapshot Distribuído, podendo assim levar a inconsistências nas leituras durante a sua execução.

Slide 15:

- O próximo protocolo MTCC que vamos abordar, é o Conservativo ou, a sua abreviatura, CON. Este, como o próprio nome indica, é o oposto do Otimista.
- Em vez de começar automaticamente a executar as funções de um DAG, este protocolo começa por acordar com todos os nós envolvidos na execução do DAG, qual o Snapshot Distribuído que irão utilizar para este efeito.
- Cada nó começa por criar, a pedido do Planeador de Funções, o seu Keyset-Overlapping Cut para as chaves que irá utilizar na execução das respetivas funções.
- De seguida, o Planeador de Funções, realiza a união de todos os Snapshots recebidos, criando assim um Snapshot Distribuído.
- No caso de alguma Cache não conter a versão de uma dada chave, a qual consta no Snapshot Distribuído, é realizado a esse mesmo nó, um pedido, para que vá buscar à Anna essa mesma versão.
- Após acordado o Snapshot Distribuído, o DAG é executado com a garantia de poder terminar com sucesso, mesmo num cenário onde a escrita é frequente. Porém, se as Caches dos nós envolvidos estivessem suficientemente sincronizadas, o acordo do Snapshot Distribuído teria apenas atrasado a execução do DAG.

Sara

Slide 16:

- O protocolo Híbrido, ou HB, combina os benefícios do Otimista com o Conservativo.
- O Otimista por correr sem coordenação é suscetível a abortar, enquanto que o conservativo nunca aborta tendo, no entanto, um grande custo de coordenação.
- Então o híbrido tenta conjugar os dois de forma a obter o melhor resultado possível.
- O híbrido começa por correr uma sub-rotina com o otimista e uma simulação do mesmo.
- Como a simulação é mais rápida que correr o otimista em si, é possível determinar mais cedo se este vai abortar.
- Assim, se a simulação abortar uma sub-rotina com o conservativo é lançado.
- Caso a simulação não aborte o Otimista continua o seu processo.

O Híbrido contém também algumas otimizações tais como:

Pré-Busca:

- Como corremos uma simulação é possível perceber que informação é necessário ir buscar à memória. Assim, este protocolo à medida que a simulação é feita notifica a cache qual informação vai ser necessária.

Abortar mais cedo:

- Quando a simulação aborta, a cache é notificada que a sub-rotina do Otimista vai terminar, impedindo a perda de computação desnecessária.

Cache de resultado das funções:

- Enquanto a sub-rotina do Otimista está em execução os resultados das funções executadas e as suas chaves são guardadas em cache.
- Caso esta sub-rotina do Otimista aborte e se inicie a sub-rotina do Conservativo as funções já executadas vão ser reexecutadas.
- No entanto se as versões das chaves presentes no Conservativo forem equivalentes às geradas pela execução do Otimista, esta nova execução não é feita e são usados os resultados das funções já presentes na cache.

- Com este protocolo concluímos o conjunto de protocolos que temos para apresentar, passemos então à discussão teórica de vários aspetos.

Slide 17:

Na discussão teórica foram tidos em consideração 4 aspetos.

Auto escalabilidade:

- O escalamento automático do sistema poderia provocar um problema de consistência.
- No entanto, em todos os protocolos mencionados as caches contém o seu próprio corte sendo que apenas as que estão envolvidas no DAG precisam de se coordenar entre si para garantir o Snapshot correto.
- Desta forma permite que a HydroCache garanta Consistência Causal Transacional, ou seja TCC, sem que se tenha de preocupar com o escalamento do sistema.

Leituras Repetidas:

- Quanto a Leituras Repetidas, todos os protocolos conseguem garantir que se 2 funções pedirem o valor de uma variável k ambas obtêm o mesmo valor.

Versões:

- Os protocolos vão criando versões temporárias enquanto estão a executar um pedido, permitindo que as outras caches tenham acesso ao valor inalterado. Estas versões temporárias são posteriormente apagadas para reduzir o impacto na memória.

Readset desconhecido:

- Quando o Readset não é conhecido, é usado inicialmente o protocolo Otimístico para explorar o Readset do DAG.
- Quando uma validação falha, é invocado o protocolo Conservativo para construir um Snapshot Distribuído de todas as chaves lidas até ter abortado.
- Desta forma, a próxima tentativa de exploração com o Otimístico não vai abortar com inconsistência em chaves já exploradas.
- Este processo é realizado até o DAG estar completamente explorado.

Após esta discussão teórica passemos agora o resultado prático.

Slide 18:

- Para esta primeira análise prática entre os protocolos foram tomadas em consideração 2 tipos de DAG, um DAG linear e um DAG em formato de V.
- Um DAG linear é um conjunto de 3 funções em que o resultado da primeira função é dado como input para a segunda e o resultado da segunda é dado como input à terceira, daí o termo linear. No entanto, num DAG com formato em V, 2 das 3 funções correm em paralelo sendo o seu output dado como input da terceira função, advindo daí o seu formato em V.
- Nos seguintes gráficos temos a latência de cada um dos protocolos tendo em conta o tipo de DAG.
- No gráfico, a barras coloridas representam a mediana do valor da latência enquanto que a barra acima desta representa o percentil 99 dos valores de latência.
- O valor mencionado por baixo do gráfico indica o valor do coeficiente dado a uma distribuição do tipo Zipfian.
- De todos os protocolos o Híbrido é que apresenta melhor resultados no conjunto de valores de mediana e de percentil 99. Estes resultados advêm do facto do híbrido tirar partido dos benefícios de 2 outros protocolos. Podendo adaptar os pontos fortes de cada um deles para cada situação que encontra.

Slide 19:

- Agora compara-se a HydroCache com outras arquiteturas, como a Anna e a ElastiCache variando o payload dado.
- É possível observar que o aumento nas outras arquiteturas é proporcional ao aumento do payload enquanto que na HydroCache esse aumento é muito menos acentuado.
- No entanto, á medida que o payload aumenta, o acréscimo do seu percentil 99 vai sendo também proporcional ao aumento do payload tal como acontece nas outras arquiteturas.
- Em geral, é possível concluir que a HydroCache melhora bastante a performance aquando comparada com arquiteturas sem cache. Para além disso a HydroCache é também a única que não apresenta inconsistências nas escritas como é mostrado na tabela à direita.

Slide 20:

- Nesta apresentação, vimos o que é um sistema FaaS, no qual analisámos em mais detalhe a implementação da HydroCache.
- Foi possível perceber que tipo de protocolos esta utiliza e quais os benefícios e desvantagens que cada um deles apresenta.
- Vimos também a potencialidade em termos de eficiência de sistemas utilizando HydroCache, comparativamente a sistemas sem cache.
- Assim podemos concluir que a HydroCache é uma boa solução para sistemas FaaS, garantindo Consistência Causal Transacional Multisite conseguindo, ainda assim, manter uma boa performance.
- Esperemos que tenham gostado, Obrigada!