# BigTable: Um sistema de armazenamento distribuído para dados estruturados (Maio 2011)

Valter Henrique de Souza Ferreira da Silva,*valter.bcc@gmail.com, UFSCar – campus Sorocaba.*

***Resumo*—Gerenciar dados distríbuidos sempre foi um grande desafio, este desafio é ainda maior quando estamos falando de terabytes de dados. O Google desenvolveu uma solução frente a este desafio, chamada BigTable, sendo um sistema de armazenamento distríbuido que gerencia dados estruturados, ela possui uma oferta e demanda de dados muito grande, em larga escala. Estamos falando de gigabytes, terabytes, petabytes de dados que são enviados para servidores comuns, sem muito desempenho em hardware. Muitos dos projetos do Google armazenam dados na BigTable, incluindo a indexação web, alguns desses projetos são mais conhecidos, tais como Google Earth, que possui milhares de imagens de satellite com um grande volume de dados, Google Finance, Google Maps, Orkut, Blogger e Google Cloud Print, tais aplicações possuem necessidades diferentes, seja em relação ao tamanho do dados que vão de URLs que direcionam para páginas web á imagens de satelite, estas aplicações possuem uma latência de requisições muito grande, desde o processamento nos servidores a entrega dos dados já processados ao clientes que o requisitou. E mesmo com essa demanda variada, a BigTable tem atendido muito bem essa demanda, fornecendo uma flexibilidade aos clientes, um alto desempenho para todos os produtos que são vinculados a ela. Neste artigo vamos discutir o modelo de dados utilizados na BigTable que oferece aos seus clientes um maior controle, um certo dinâmismo, sobre como os dados estão organizados, o projeto e implementação da BigTable.**

***Palavras-chaves***: sistema, distribuído, dados, estruturados, armazenamento.

## I. Introdução

A BigTable teve origem por meados de 2004, ela foi projetada para armazenar dados em larga escala (petabytes de dados) e ser confiável, aonde ela iria rodar em cima de milhares de servidores, ela atingiu grandes feitos, tais como : vasta aplicabilidade, aonde pode agregar vários produtos do Google na mesma, há mais de 60 projetos, produtos que utilizam a BigTable.

Os cluster da BigTable são utilizados por estes produtos abrangem uma gama de configurações, de centenas para milhares de servidores e armazenar milhares de terabtytes de dados. A BigTable, de muitas formas assemelha-se a um baco de dados, ela compartilha muitas estratégias de implementação com os bancos de dados, utilizando conceitos desde banco de dados paralelos [1] a banco de dados de memória principal [2], com isso conseguiu escalabilidade e alto desempenho.

A BigTable fornece uma interface diferentes dos sistemas citados acima. Ela não suporta modelo de dados relacional, na verdade ela fornece ao seu cliente um modelo de dados simples, que este por sua vez irá fornecer um dinamismo maior sobre o controle dos dados e o formato que os mesmos se encontram, permitindo aos clientes pensar sobre as propriedades de localização dos dados representados no armazenamento subjacente.

Os dados são indexados utilizando nomes de linhas e colunas que podem ser arbitrários. A BigTable trata os dados como strings sem interpretá-las, embora que os clientes frequentemente convertem este dados de diversas formas em dados estruturados ou semi-estruturados em strings.

Os clientes podem controlar a localização dos dados através de escolhas feitas em seus respectivos esquemas, a BigTable possui um esquema possui parâmetros que permitem aos clientes controlarem dinamicamente aonde será entregue os dados, em memória ou em disco.

## II. Modelo de dados

Por ser projetada para ser distribuída, a BigTable é esparsa, é mapeada em várias dimensões e ordenada. O mapeamento é indexado por uma linha, coluna e um tempo (timestamp), cada valor no mapeamento é um vetor de bytes sem interpretação.

(Linha: String; Coluna: String; Tempo:int64) : String

Na figura abaixo será demonstrado um exemplo do modelo de dados da BigTable.

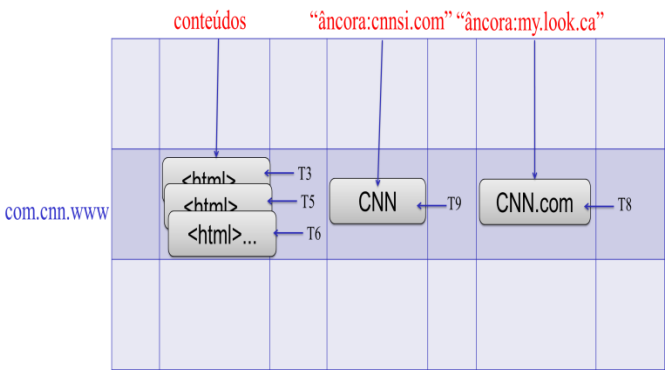

Figura 1: Um exemplo de uma tabela que armazena páginas web. O nome da linha é a url invertida, a família, conjunto de colunas são chamados de família, a célula que contém o valor se chama *conteúdo*, neste caso são os conteúdo de uma página web. E a família de colunas chamada â*ncora* contém o texto para qualquer ancora na página referenciada. A página home da CNN é referenciada por outras páginas web que neste caso são Sport Ilustrated e My-Look. Cada célula âncora tem versões, a coluna *conteúdo* possui três versões, nos tempos (T3, T5 e T6).

Este modelo de dados foi definido depois dos engenheiros do Google examinarem vários sistemas que teriam o mesmo comportamento que a BigTable e levaram em consideração a grande variedade de uso que seria feita da mesma. Utilizando um exemplo mais sólido que possa explicar um pouco melhor o porque deste modelo de dados, vamos supor que queremos manter a copia de uma coleção muito grande de páginas web e as informações relacionais destas páginas possam estar em diferentes projetos, vamos denominar esta tabela por *TabelaWeb* aonde nela teríamos URLs como chave de linha e alguns aspectos de páginas web como nome da coluna, e armazenaríamos o conteúdo destas páginas em colunas chamadas *conteúdo*.

## LINHAS

As chaves em uma linha são *strings* arbitrárias (atualmente possuem o tamanho maior que 64 KB, mas para a maioria dos usuários, as strings tem tamanho de 10-100 bytes). Cada leitura, escrita em uma linha é atômica, isso independe do número de colunas diferentes que estão sendo lidas ou escritas na linha, isso faz com que fique mais fácil para os clientes pensarem sobre o comportamento de atualizações concorrentes na mesma linha.

Os dados são mantidos juntos por ordem lexicografica, isso pela chave linha.

O tamanho do intervalo de uma tabela é dinamicamente particionado, cada intervalo de linhas é chamado de *tablet*, o qual é uma unidade de distribuição e balanceamento de carga. Com isso, as leituras em um intervalo são eficientes e tipicamente requerem a comunicação com poucas máquinas.

Os clientes podem explorar esta propriedade por selecionar as linhas chave, assim eles podem obter uma boa localização para acessos dos seus respectivos dados. Por exemplo, na *TabelaWeb* páginas em um mesmo domínio são agrupadas juntas em linhas contiguas, por inverter os componentes do hostname contido nas URLs. Por exemplo nós armazenamos o dados para *maps.google.com/index.html* na linha chave *com.google.www/index.html*. Este tipo de abordagem aonde se armazena a página do mesmo domínio próximo da outra faz com que a análise de domínio e host seja mais eficiente.

## FAMÍLIA DE COLUNAS

O agrupamento de colunas chaves são chamados de *família de colunas*, que formam uma unidade básica de controle de acesso. Para todo dado que é armazenado em uma família de coluna geralmente é do mesmo tipo (para que a família de colunas possa ser compressadas juntas). A família de colunas deve ser criada antes dos dados serem armazenados sobre qualquer coluna que pertença aquela família, depois da família criada qualquer coluna chave daquela família já pode ser utilizada. Este tipo de comportamento foi intencional para que seja pequeno o numero de famílias colunas, e que as famílias raramente mudem de operação, em contra partida uma tabela pode ter um número ilimitados de colunas.

A coluna chave é nomeada utilizando a seguintes sintaxe:*família:qualificador*. O nome da coluna família devem ser capazes de serem expressos em impressão, os qualificadores podem ser strings arbitrárias. Um exemplo de família de colunas no nosso exemplo da *TabelaWeb* é a língua, aonde guarda-se em que língua determinada página web foi escrita. Nós utilizamos apenas uma colunas chave na família de colunas, e armazenamos o id da linguagem em cada página web. Uma outra família de coluna muito útil também seria a coluna *âncora*, como exemplificado na figura 1. O qualificador é o nome do site referenciado, a célula *conteúdo* é o link em texto.

Controle de acesso em disco e memória são fatores que contam no desempenho em nível da família de colunas. No exemplo *TabelaWeb*, estes controles permitem gerenciar muitos dos diferentes tipos de aplicações, alguns que adicionam novas base de dados, alguns que leem a base de dados e criam diferentes famílias de colunas e algunas apenas são somente permitidos para visualizar o dado, há alguns casos que não é possível visualizar todas as colunas devido as políticas que são estabelecidas na tabela.

## TEMPOS (*TIMESTAMPS*)

Cada célula na BigTable pode conter múltiplas versões do mesmo dado, estas versões são indexadas pelos tempos em que foram inseridas. Este atributo *tempo (timestamp)* é um inteiro de 64 bits. Eles podem atribuídos pela BigTable, na qual eles representam o momento atual, em microssegundos, ou podem ser especificados pelas aplicações clientes também.

```
1. // Abrir a tabela
2. Table *T = OpenOrDie("/bigtable/web/webtable");
3.
4. // Escrever uma nova âncora e apagar a antiga
5. RowMutation r1(T, "com.cnn.www");
6. r1.Set("anchor:www.c-span.org", "CNN");
7. r1.Delete("anchor:www.abc.com");
8. Operation op;
9. Apply(&op, &r1);
```

Figura 2: Escrevendo na BigTable

Aplicações que precisam evitar colisões devem gerar um

*timestamp* único. Diferentes versões de uma célula são armazenadas em *timestamps* ordenados decrescentemente, isso para que as versões mais recentes possam ser lidas primeiro.

Para não dificultar no versionamento dos dados, há a opção de configurar em uma coluna de configuração na família de tabelas, aonde a BigTable utiliza esta coluna para saber se utilizará o coletor de lixo (*garbage collect)* para apagar versões automaticamente. O cliente pode especificar também que apenas as últimas n versões de uma célula devem ser mantidas, ou que apenas as versões mais novas devem ser mantidas, por exemplo, manter os valores que foram escritos nos últimos 7 dias.

No nosso exemplo, na *TabelaWeb*, podemos configurar os *timestamps* das páginas que foram rastreadas armazenadas nos conteúdos, uma coluna que contém os tempos em que as páginas foram realmente rastreadas, pesquisadas. A coleta de lixo descreve que apenas as três versões mais recentes devem ser mantidas apenas, de todas as páginas, por exemplo.

## III. API

A API da BigTable possui funções para criar, apagar tabelas e famílias de colunas, ela também fornece função para mudança de cluster, tabelas e metadados de família de coluna, tais como controles dos direitos de acesso.

As aplicações clientes podem escrever ou apagar os valores na BigTable, pesquisar valores para linhas individuais, ou iterar sobre um conjuntos de dados na tabela.

A figura 2 mostra uma código em C++ que utiliza a função *RowMutation* que é uma abstração para realizar um conjunto de atualizações. A chamada em *Apply* realiza uma mutação atômica para a *TabelaWeb*, ela adiciona a âncora to www.cnn.com e apaga a âncora diferente.

A figura 3 exibe um código desenvolvido em C++ que utilizar um *Scanner* que nada mais é que uma abstração para iterar sobre todos os âncoras em uma linha em particular.

Os clientes conseguem iterar com em cima de diversas colunas de famílias, e há vários mecanismo para limitar colunas, linhas e tempos produzidos por uma leitura. Por exemplo, nós podemos restringir a leitura acima para apenas produzir âncoras naquelas cujas colunas correspondem a expressão regular:*.cnn.com, ou apenas produzir âncoras cujo os tempos caem dentre de 10 dias á partir do tempo atual.

```
1. Scanner scanner(T);
2. ScanStream *stream;
3. stream = scanner.FetchColumnFamily("anchor");
4. stream->SetReturnAllVersions();
5. scanner.Lookup("com.cnn.www");
6. for (; !stream->Done(); stream->Next()) {
7. printf("%s %s %lld %s\n",
8. scanner.RowName(),
9. stream->ColumnName(),
10. stream->MicroTimestamp(),
11. stream->Value());
12. }
```

Figura 3: Lendo da BigTable

Dentre as várias características da *BigTable* ela permite que o usuário manipular os dados de forma bastante complexa. Ela suporta primeiramente transações de uma única linha, a qual pode ser utilizada para realizar operações de leitura, modificação e escrita atomicamente nos dados armazenados em uma única linha. Ela não suporta transações de modo geral através entre as linhas chaves, entretanto ela fornece uma interface para lotes de escritas através das linhas chaves nos clientes. Ela permite que células são utilizadas como contadores de inteiros. E por último a BigTable suporta a execução de scripts fornecidos pelos clientes em um espaço de endereços dos servidores. Os scripts são processados em uma linguagem desenvolvida pela própria Google, esta linguagem se chama Sawzall [3].

A BigTable pode ser utilizada juntamente com *MapRecude* [4], aonde este por sua vez é um framework que é executado em larga escala utilizando a computação paralela desenvolvida no Google.

## IV. BLOCOS DE CONSTRUÇÃO

A construção de uma ferramenta tão grande como a BigTable utiliza algumas outras ferramentas e infraestrutura do Google.

Os componentes que a BigTable utiliza o Google File System mais conhecido pela sigla GFS [4] para armazenar log e arquivos dos dados. Um cluster da BigTable tipicamente opera sobre uma *piscina (pool)* de máquinas que rodam uma variedade muito grande de outras aplicações distribuídas, e os processos da BigTable são compartilhados na mesma máquinas com processos com outras aplicações. A BigTable depende de uma gerência de cluster para agendar os trabalhos, gerenciar os recursos que são compartilhados em uma mesma máquina, lidando com falhas de máquinas e monitoramento do estado da máquina.

*SSTable* é um formato que é utilizado internamente para armazenar os dados da BigTable. Um *SSTable* fornece um mapeamento imutável de chaves e valores, aonde ambas as chaves e valores são *string* de bytes arbitrários. São fornecidos operações para pesquisar por valores associados a uma chave específica. E para iterar sobre todos os pares chaves | valores

em um intervalo específico. Internamente, cada *SSTable* contém uma sequência de blocos, cada bloco possui o tamanho de 64 KB, mas isso é configurável). Um índice de bloco é armazenado no final da *SSTable*, este índice é utilizado para localizar blocos, o índice é carregado em memória quando a *SSTable* é aberta. Uma pesquisa pode ser realizada com uma única busca em disco, nós primeiro encontramos o bloco apropriado por realizar uma busca binária em uma pesquisa nos índices da memória. Opcionalmente uma *SSTable* pode ser completamente mapeada em memória, a qual permite que nós pesquisamos e buscamos sem ao menos tocar no disco. BigTable baseia-se em alta disponibilidade e bloqueio de serviço persistência distribuída chamados *Chubby*[5].

Um *Chubby* consiste de cinco replicas ativas, uma a qual e eleita para ser o mestre and ativamente atendendo as requisições que lhe são enviadas, o serviço esta vivo quando a maioria das réplicas estão rodando e podem se comunicar umas com as outras. *Chubby* utiliza o algoritmo *Paxos* [6] para manter as suas réplicas consistentes e lidar com falhas. *Chubby* fornece um espaço de nomes que consistem de diretórios e pequenos arquivos. Cada diretório ou arquivo pode ser utilizado como um bloqueio, e leituras e escritas a um arquivo são atômicas. A biblioteca cliente *Chubby* fornece cacheamento consistente dos arquivos *Chubby*.

Cada cliente *Chubby* mantém uma sessão com o serviço *Chubby*. Uma sessão cliente expira se a renovação não for possível . Quando o cliente expira ele perde todos os bloqueios e mecanismos de abertura. Clientes *Chubby* também podem registrar *call-backs* no arquivos e diretórios *Chubby*, isso ocorre para notificações de mudanças ou expiração de sessão.

BigTable utiliza o *Chubby* para uma variedade de tarefas para garantir que há mais de um servidor ativo em qualquer momento, para armazenar os dados de localização de boot da BigTable, para que se possa descobrir servidores de *tablets* e finalizar servidores de tablets também. Para armazenar o esquema de informações da BigTable (a família de colunas informação para cada tabela) e para armazenar a lista de controle de acesso. Se *Chubby* se torna indisponível por um longo período, a BigTable se torna indisponível.

## V. Implementação

A implementação da BigTable tem três componentes principais: a biblioteca que esta ligada a cada cliente, um servidor mestre e por fim muitos servidores de tablets.
Servidores de *tablets* podem ser adicionados ou removidos dinamicamente dos *cluster* para adaptar as mudanças em cargas de trabalho.
O mestre é responsável por atribuir *tablets* aos servidores de *tablets*, detectando a adição e expiração de *tablets* em servidores, balanceamento de carga do servidor de *tablet*, e coletor de lixo nos arquivos do GFS.
Além disso lida com mudanças no esquema tais como tabelas e criações de famílias de colunas. Cada servidor de *tablet* gerencia um conjunto de tablets, tipicamente nós temos algo entre dez e milhares de tablets por servidor de tablet..
O servidor de *tablet* lida com as requisições de leitura e escrita dos *tablets* que já estão carregados e também dividem os *tablets* a medida com que crescem muito.
Como muitos sistemas distribuídos possuem um único servidor, os clientes dos dados não se comunicam com o mestre, mas sim com o servidor de tablete, essa comunicação evita-se sobrecarregar a comunicação do mestre com os demais servidores de *tablet*, ele se comunica diretamente com o servidor de tablets para leitura e escrita. Isso porque os clientes BigTable não se baseiam no mestre para localização do tablete, a maioria dos clientes nunca se comunicam com os seus mestres. Como resultado o mestre é rapidamente carregado.
Um cluster da BigTable armazenada um número de tabelas, cada tabela consiste em um conjunto de tablets, e cada tablete contem todos os dados associados em uma linha. Inicialmente, cada tabela consiste de apenas um tablete. Como a tabela cresce e elea automaticamente se divide em múltiplos tablets, cada um aproximadamente de 100 á 200 MB de tamanho por padrão.

## VI. Localização do Tablet

Para determinar a localização do tablet usa-se uma hierarquia de 3 níveis para armazenar as informações do *tablet.*

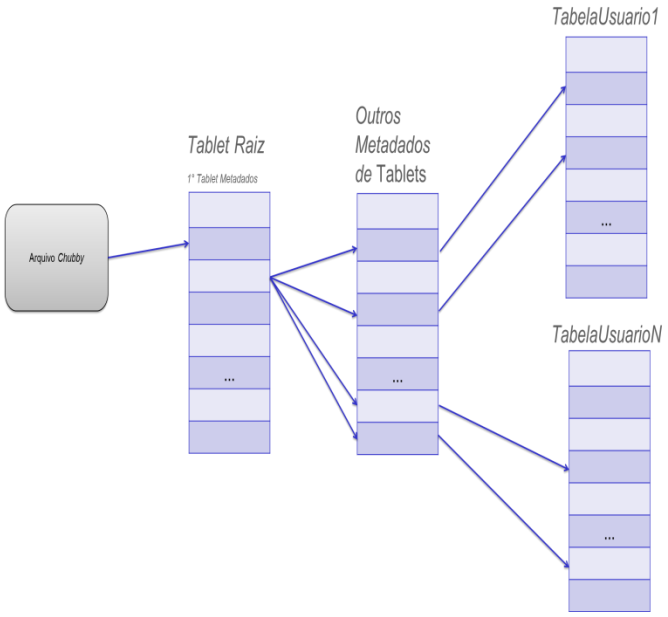


Figura 4: Hierarquia da localização do *tablet*

O primeiro nível é aonde se armazena o arquivo *Chubby* que contém a localização do *tablete* raiz. O *tablet* raiz por sua vez a localização de todos os *tablets* em uma tabela especial, METADADOS.
Cada *tablet* METADADOS contém a localização do conjunto de *tablets* do usuário. O *tablet* raiz é apenas o primeiro *tablete* que contém os metadaos, porém é tratada de forma especial,

ela nunca se divide, isso para garantir que a hierarquia não tenha mais do que três níveis. As tabelas METADADOS armazenam a localização dos *tablets* em uma linha chave que é uma codificação dos identificadores da tabela *tablets* e seus finais de linha.

Cada linha METADADO armazena aproximadamente 1KB de dados na memória. Com um limite de 128MB de metadados sobre *tablets*, a hierarquia de três níveis é o suficiente para endereçar 234 *tablets*(ou 261 bytes em 128 MB *tablets*).

A biblioteca do cliente realiza o cache da localização dos tablets. Se os clientes não sabem a localização do tablete, ou se descobrem que a informação do cache esta incorreta então ele recursivamente procura a localização correta na hierarquia. Se o cache do cliente esta vazio, o algoritmo de localização requer três tentativas na rede, incluindo uma para leitura do *Chubby*. Se o cache do cliente esta velho, o alogoritmo de localização pode necessitar de 6 tentativas, por as entradas do cache antigo só são descobertar dado o desaparecimento de algum *tablet*, os *tablets* de metadados não mudam de local com frequência. Entretanto os tablets de localização são armazenados em memória, então não há necessidade de acessar o GFS, esse acesso desnecessário ao GFS se tornou possível a operação de *prefetch* nas bibliotecas clientes, aonde se realiza o *prefecth* das localizações dos *tablets*, aonde se realiza a leitura dos metadados para mais de um *tablete* quando realiza a leitura na tabela metadado. É armazenado também uma informação secundária na tabela metadados, incluindo o log de todos os eventos pertencentes a cada *tablet* . Essa informação é de grande ajuda para realizar *debugging* e análise de desempenho.

## VII. Atribuição do Tablet

Cada *tablet* é atribuído a um serividor *tablet* de cada vez. O mesmo mantém rastreado o conjunto de servidores de *tablet* operantes, e a atribuição atual de *tablets* para o servidor de *tablets*, incluindo quais *tablets* não estão atribuídos. Quando um *tablet* não tem nenhuma atribuição, e o servidor de *tablet* tem espaço disponível, o mestre designa o *tablet* ao servidor enviando uma requisição a ele.
A BigTable utiliza o arquivo *Chubby* para manter o rastreamento dos servidores de tablets. Quando um servidor de *tablet* inicia, ele cria e adquire um bloqueio exclusivo, o qual é único em um diretório específico do *Chubby*. O mestre fica monitorando este diretório para descobrir novos servidores de tablets. Enquanto o mestre não perder o bloqueio exclusivo que lhe foi atribuído ele continua servindo, ele tentará adquirir novamento este bloqueio exclusivo enquanto ele estiver operante. Se o arquivo não existir mais então o servidor de *tablet* não vai ser disponível mais, então ele se anula.
A responsabilidade de mestre é de detectar quando um servidor de tablet não esta servindo mais os seus tablets e atribuir novamente os tablets assim que possível. Detectar quando um servidor de tablet não esta servindo mais tablets, o mestre periodicamente pergunta para cada servidor de tablet o estado do seu respectivo bloqueio.Se o servidor de tablet responder que ele perdeu o bloqueio, ou se o mestre for incapaz de alcançar o servidor durantes suas últimas tentativas, o mesmo tenta adquirir um bloqueio exclusivo aos servidores. Se o mestre for capaz de adquirir o bloqueio, o Chubby esta operante e o servidor de tablet não esta, ou tendo problemas para alcançar o Chubby, então com isso o mestre tem certeza que o servidor de tablet não pode servir novamente, como isso ele apagar o seu arquivo servidor. Uma vez que o servidor de arquivo foi apagado, o mesmo pode mover todos os tablet que estavam anteriormente atribuídos ao servidor em um conjunto de tablets não atribuídos. Para garantir que o cluster da BigTable não é vulnerável aos problemas de rede entre o mestre e o Chubby, o mestre anula a si mesmo se a sessão do Chubby expirar. Entretanto como descrito acima, a falha do mestre não mudar a atribuição do tablets aos servidores de tablets.

Quando é iniciado o mestre pelo sistema gerenciador de cluster, ele precisar descobrir a atual atribuição de tablets antes que ele possa realizar as mudanças necessárias. O mestre executa os seguintes passos ao iniciar, o mestre obtém um bloqueio mestre para o Chubby, o que evitar a atuais instâncias mestres, depois o mestre pesquisa nos diretórios do Chubby por servidores operantes, com isso ele poe se comunicar com estes servidores para descobrir quais tablets cada uma deles possui, o mestre pesquisa então na tabela metadados para aprender sobre o conjunto de tablet atual.

Independente do que for encontrado nesta pesquisa, seja um tablet não atribuído a algum servidor de tablet, o mestre adiciona este tablet não atribuído a um conjunto de tablet não atribuídos.

O conjunto de tablet muda somente quando uma tabela é criada ou apagada, duas tablets existentes que são unificadas para formar um tablet maior ou dividir um tablet maior em dois menores.

As divisões das tablets são tratadas especialemente desde que elas são iniciadas pelo servidor de tablet, então o servidor realiza a divisão, gravando a informação para uma nova tablet na tabela metadados. Quando a divisão acontece ele notifica o mestre. Em caso da notificação de divisão seja perdida porque o servidor mestre tenha caído, o mestre detecta uma nova tablet quando pergunta ao servidor de tablet para carregar o tablet que acabou de ser dividido. O servidor de tablet vai notificar o mestre da divisão porque a entrada de registro da divisão do tablet se encontra na tabela metadado, que esta por sua vez irá apenas espeficar a porção do tablet que o mestre pediu para ser carregado.

## VIII. Servindo tablet

O tablet possui um estado persistente que é armazenado no GFS, como ilustrado na figura 5.

Figura 5: Representação de um tablet

As atualizações são enviadas para um log que armazenam as operações realizadas. Dessas atualizações, as mais recentes são enviadas e armazenadas em memória aonde é ordenada em buffer chamado memtable, as atualizações mais antigas são armazenas na sequencia dos SSTables. Para recuperar um tablet, o servidor de tablet ler os metadados deste tablet da tabela metadados. Este metadado contém a lista de SSTables que abrangem um tablet e um conjunto de pontos de retorno, estes pontos de retorno contém dadaos para o tablet. O servidor lê os índices das SSTable na memória e reconstrói a memtable aplicando todas atualizações que foram submetidas desde os pontos de retorno.
Quando uma operação de escrita chega em um tablet servidor, ele verifica se a escrita é valida, então verifica se quem enviou esta autorizado a tal operação. A autorização é realizada lendo uma lista de usuários habilitados a escrever, ele lê estes dados de um arquivo Chubby, este arquivo esta na maioria das vezes em cache no próprio cliente.
Depois que a escrita foi submetida o seu conteúdo é inserido na memtable. Quando a operação de leitura chega ao servidor , ele realiza uma operação similar, de verificação de autorização para leitura.
Leituras e escritas podem continuar a ocorrer enquanto as tablets podem ser divididas e unificadas.

## IX. Compactação

Como dito acima, as operações de escritas vão para a memtable, com isso ela aumenta de tamanho, quando o limite é atingido a memtable fica estática, é uma nova memtable é criada, e a memtable é convertida para uma SSTable e escrita no GFS. Esta é a compactação menor, que tem dois objetivos, uma de encolher a memória utilizada dos servidores de tablets, e reduzir a quantidade de dados que tem que ser lidos caso o servidor caia. As leituras e escritas que são enviadas podem continuar normalmente enquanto puder ser realizado as compactações. Toda compactação menor cria uma nova SSTable. A compartação de união lê os conteúdo de algumas SSTable e as memtable, e escreve em uma nova SSTable. A entrada de SSTable e memtable pode ser descartadas assim que a compactação for terminada. Uma compactação que re-escreve todas as SSTable em uma única SSTable é chamada de compactação maior.

## X. Refinamentos

A implementação descrita nas seções anteriores requem um número de refinamentos para atingir o alto desempenho esperado, disponibilidades e confiabilidade exigidas pelos usuários. Esta seção descreve algumas destas implementações em maiores detalhes.

### Grupos Locais

Clientes podem agrupar várias famílias de colunas em um grupo local, uma SSTable é criada para cada grupo local em cada tablet, separando as famílias de colunas que não são acessadas juntamentes, permitindo assim uma maior eficiência na leitura. Por exemplo, na TabelaWeb, imagine que temos língua e checksums que podem ser um grupo local, e o conteúdo da página pode ser em um grupo diferente, uma aplicação que desejar ler os metadados não precisa ler todos os conteúdos das páginas.

### Compressão

Clientes podem ou não controlar quais SSTables de um grupo local serão compressados, e qual o formato da compressão, esse formato é especificado pelo cliente. A perda de espaço por causa da compressão em cada bloco separadamente ocorre, mas isso é intencional pois por causa disso a uma vantagem de poder ler as SSTables sem haver a necessidade de descompactar os blocos.
Há dois tipos de compressões, uma chamada Bentley and McIlroy o qual compressa longas strings em comuns através de uma janela muito larga. O segundo tipo é a compressão rápida que olha por repetições em uma janela pequena de 16 KB de dados. Ambas as compressões são muito rápidas tanto para codificar quanto para decodificar, elas codificam em média de 100 MB á 200 MB por segundo, e decodificam a 400 MB á 1000 MB por segundos, na máquinas mais atuais. Como o projeto requer um processamento muito rápido por causa dos grandes volumes de dados, a Google optou por velocidade ao invés focar em espaço.

### Cache para desempenho

Todos os refinamentos são focados para aumentar a velocidade com que a BigTable executa as requisições enviadas a ela, uma forma é realizar o cache nos servidores de tablets que são divididos em dois níveis.
O cache gerado pela busca é um cacheamento de maior nível que efetua o cache em pares de chaves retornados pela interface de código das SSTable. O cache gerado por blocos é de menor nível que efetua o cache em blocos que são lidos do GFS. O cache de pesquisa é mais útil para aplicações que tem o costume de ler o mesmo dado repetidamente. O cache em bloco é mais útil para aplicações que tem o costume de ler os dados que são próximos dos dados que acabaram de ler.

### LUPAS DE FILTRO

Apesar de um nome nada comum, as lupas de filtros tem por definição diminuir o número de acessos ao disco caso alguma SSTable não esteja em memória, já que se a SSTable não estiver em memória temos que busca-la no disco, a lupa de filtro pode ser especificada pelo cliente, que especifica em quais grupos locais das SSTable que será criado a lupa de filtro. Uma lupa de filtro permite pesquisar se uma determinada SSTable possui algum dado em um par linha, coluna específico, isso faz com que se reduze bastante, em algumas aplicações, o acesso ao disco, no caso de não ser necessário acessar o disco para um dado que não existe mais.

### IMPLEMENTAÇÃO ENVIO DE LOG

A BigTable é um sistema em larga escala e distribuído, imagine criar um log para todos os tablets separadamente, seria criar um número muito grande de arquivos que precisariam ser escritos concorrentemente no GFS, o que acarretaria em grande acúmulo de dados e trafego na rede desnecessários. A alternativa que foi desenvolvida foi concatenar as mudanças em um único log por servidor tablet, este sim seria enviado ao GFS. Utilizando apenas um log faz com que seja fornecido um desempenho significativo durante as operações normais, mas pode dificultar a recuperação das máquinas que apresentam falhas.

Quando um servidor de tablet falhar ou morrer, os tablets que ele serve serão deslocados para um novo servidor tablet. Para recuperar o estado para um tablet, o novo servidor de tablet precisa reaplicar as mudanças nestes novos tablets, isso com base no log enviado pelo servidor de tablet que apresentou falha ou morreu.Uma abordagem seria para cada novo servidor tablet seria ler este log completo e aplicar apenas os registros necessários para que os tablets sejam recuperados. Entretanto nesta abordagem, se 100 máquinas estivessem atribuídas ao mesmo tablet de um servidor de tablet que falhou, então o log seria lido 100 vezes, uma vez por cada servidor. O que é feito para evitar a leitura duplicada é ordenar as entradas do log na seguinte ordem das chaves: (tabelas, nome da linha, número da sequência do log). Em uma saída ordenada todas as mudanças para um tablet em particular é contiguo e portanto a leitura se torna mais eficiente com um único acesso ao disco já que basta que a leitura seja em sequência. Para paralelizar a ordenação, particiona-se o log em segmentos de 64 MB, e ordenada cada segmento em paralelo nos servidores tablets diferentes.

Este processo de ordenação é coordenado pelo mestre e iniciado quando o servidor de tablet indica que precisa recuperar as mudanças de algum log que lhe foi enviado.

Alguns picos são causados no GFS devido as escritas dos logs , para evitar este picos cada servidor de tablet possui duas threads, uma para escrever o seu próprio log, e apenas uma delas é ativada por vez. Se a escrita esta ativa e o desempenho cai então ele é deslocado para outra thread, e as mudanças que esta em

Alguns picos são causados no GFS devido as escritas dos logs , para evitar este picos cada servidor de tablet possui duas threads, uma para escrever o seu próprio log, e apenas uma delas é ativada por vez. Se a escrita esta ativa tendo desempenho baixo então a thread é deslocada para outra thread, e as mudanças que estavam sendo executadas são transferidas para essa nova thread. Entradas dos logs contém números sequenciais permitem fazer a recuperação, como citado anteriormente, evitando assim as entradas duplicadas.

### ACELERANDO A RECUPERAÇÃO DE TABLET

Quando o mestre desloca um tablet de um servidor para outro, o servidor fonte realiza uma compactação menor daquele tablet. Esta compactação reduz o tempo de recuperação daquele tablet reduzindo a quantidade de estados não compactados no log do servidor de tablet. Ao acabar esta compactação, o servidor de tablet para de servir o tablet, na verdade ele descarrega o tablet do servidor, o servidor de tablet realiza outra compactação menor de forma que elimine qualquer estado não compactado no log do servidor de tablet enquanto a primeira compactação menor estava sendo realizada. Após esta segunda compactação menor estar completada o tablet pode ser carregado em outro servidor tablet sem precisar de recuperação atráves das entradas do log.

## XI. AVALIAÇÃO DE DESEMPENHO

O cluster da BigTable possui vários servidores de tablets que ajudam a quantificiar o desempenho e escalabilidade da BigTable, este número de servidores é variado. Os servidores de tablet estão configurados para usar 1 GB de memória para escrever no GFS com célula consistingdo de 1786 máquina com dois discos rígidos de 400 GB cada um deles.

N máquinas clientes geradadas gera a carga da BigTable usada neste teste. Foi utilizado o mesmo número de clientes como o número de servidores de tablet para garantir que o cliente não tenha algum gargalo. Cada máquina possui dois core, *dual core* Opteron 2GHz, o necessário para a memória física trabalhar rodando todos os processos, e um link gigabit.

O arranjo das máquinas é em dois níveis de rede em forma de árvore ligado com cerca de 100-200 Gbps de largura de banda total disponível na raiz. As máquinas, todas elas, estão no mesmo prédio and portanto o tempo de viagem entre qualquer par de máquinas era menor que milisegundos.

O mestre e o servidor de tablet, os clientes de teste e os servidores do GFS todos rodam em um mesmo conjunto de máquinas, cada máquina roda um servidor do GFS, algumas das máquinas também rodavam os servidores de tablet, ou processos dos cliente ou processos de outras execução que estavam sendo utilizadas no *pool* naquele mesmo momento do experimento. L é o número de chaves linha na BigTable envolvida no test, L era escolhida então o *benchmark* lia ou escrita aproximadamente 1 GB de dados por servidor tablet.

A escrita em sequencia utiliza chaves de nomes de 0 a 1. Neste espaço de linhas chaves estão particionadas em 10 partes iguais com o mesmo intervalo. Os intervalos eram atribuídos a vários clientes pela central de agendamento que atribui o próximo intervalo disponível ao cliente assim que o cliente terminasse de processar o intervalo anterior atribuído a ele.

Com essa atribuição dinâmico a melhor de migrar os efeitos de variações de desempenho causadas pelo execução de máquinas clientes. Foi escrito uma única string embaixo de cada linha, cada string foi gerada de forma aleatória e era portanto incompreensível, além, as strings das linhs chaves eram distintas, então tentam cruzar os dados destas string não era possível. A leitura sequencial era gerada da mesma forma que foi gerada a escrita, mas em vez de a escrita ser feita em baixo da linha chave, ele lê a string armazenada abaixo da linha chaev, a qual já foi escrita anteriormente.

A pesquisa no *benchmark* é similar a leitura sequencial, mas utiliza um suporte fornecido pela API BigTable para pesquisar por todos os valores em um intervalo de linhas.

Utilizando a pesquisa reduz o número de RPC executadas pelo *benchmark* desde que uma única busca RPC traz uma sequencia enorme de valores de um servidor de tablet.

A figura 6 e 7 mostram duas visões de desempenho dos *benchmarks* quando são executadas leituras e escritas de 1000 bytes de valores por segundos por servidor de tablet, o gráfico mostra o conjunto de operações realizadas por segundo.

| Experimento | Quantidade de servidores de tablets | | | |
|---|---|---|---|---|
| | 1 | 50 | 250 | 500 |
| Leituras aletórias | 1212 | 593 | 479 | 241 |
| Leituras aletórias (mem) | 10811 | 8511 | 8000 | 6250 |
| Escritas aleatórias | 8850 | 3745 | 3425 | 2000 |
| Leituras sequênciais | 4425 | 2463 | 2625 | 2469 |
| Escritas sequênciais | 8547 | 3623 | 2451 | 1905 |
| Pesquisas | 15385 | 10526 | 9524 | 7843 |

Figura 6: Tabela de taxa de número de 1000 bytes escritos e lidos por segundo por servidor de tablet.

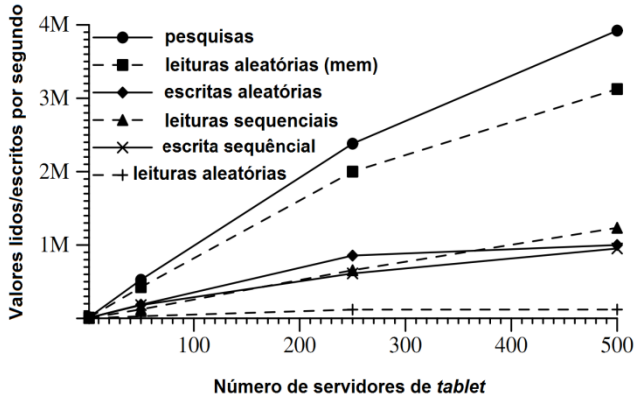


Figura 7 : Gráfico de taxa de número de 1000 bytes escritos e lidos por segundo por servidor de tablet.

### DIMENSIONAMENTO

O agrupamento faz com que a taxa de saída aumente bastante, como pode-se ver nas figuras 6 e 7, de 1 para 500 servidores de tablet.

| Quantidade de servidores de tablet | Quantidade de clusters |
|---|---|
| 0 ... 19 | 259 |
| 20 ... 49 | 47 |
| 50 ... 99 | 20 |
| 100 ... 499 | 50 |
| ➢ 500 | 12 |

Figura 8: Distribuição de número de servidores de tablet nos clusters da BigTable.

Por exemplo, o desempenho de leituras aleatórias da memória aumenta em um fator 300 enquanto o número de servidores tablet aumentam por um fator de 500. Este comportamento acontece porque o gargalo acontece na CPU do servidor de tablet. Entretando, o desempenho não aumenta linearmente, para a maioria dos *benchmarks*, há uma queda significante de saída por sevidor quando vai de 1 para 50 o número de servidores de tablet. Essa queda acontece por causa do não balanceamento de carga nas configurações dos vários servidores, frequentemente ocorre por causa dos processos contidos na CPU e na rede.

O algoritmo de balanceamento de carga tenta lidar com este não balancemanto, mas não consegue fazer um balanceamento perfeito por causa de duas razões principais, uma seria o rebalanceamento prejudica o número movimentos de tablet, sendo que um tablet fica indisponível quando esta sendo deslocado ou rearranjado, algo em torno de 1 segundo, e a carga gerada pelos *benchmarks* desloca em torno de como o valor de referência avança.

A leitura aleatória pelo *benchmark* mostra que é o pior dimensionamento, este comportamento ocorre porque foi transferido blocos grandes de 64 KB sobre uma rede que lê 1000 bytes. Esta transferência é saturada compartilhando links de 1 GB na rede e como resultado a taxa de saída caiu significamente conforme aumenta-se o número de máquinas.

## XII. CONCLUSÃO

Conclui-se que a BigTable se tornou uma ferramenta fundamental em algumas aplicações do Google como Orkut, Google Maps, Blogger e Google Earth, aonde consegue atingir escalabilidade e confiabilidade de maneira simples e eficiente utilizando outras ferramentas como o Google File System, Zippy entre outros. Visando sempre ocupar o menor espaço possível mas focando ainda mais no desempenho, na velocidade das suas aplicações. A BigTable se mostrou uma ótima ferramenta distribuída que foi construída com o auxílio de outras já desenvolvidas pela Google.

## REFERÊNCIAS

[1] ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. C. Integrating compression and execution in column-oriented database systems. Proc. of SIGMOD (2006).

[2] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SK-OUNAKIS, M. Weaving relations for cache performance. In The VLDB Journal (2001), pp. 169– 180.

[3] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In Proc. of the 3rd OSDI (Feb.1999), pp. 45– 58.

[4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND, G. P., AND WILSON, W. G. DB2 parallel edition. IBM Systems Journal 34, 2 (1995), 292– 322.

[5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In Proc. of the 1st NSDI (Mar. 2004), pp. 253– 266.

[6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In Data Compression Conference (1999), pp. 287– 295.