باسمه تعالی

جزوه درس کلان داده

استاد: دکتر محمد امین صادقی

تهیه کننده: رضا سعیدی نیا

تقدیم به روح پدرم حاجی نورالله سعیدی نیا

روحش شاد و یادش گرامی

برای شادی روح همه درگذشتگان صلوات

فصل اول: مقدمه (چرا کلان داده؟)

چکیده: اصطلاح کلان داده[1] به خاطر انفجار افزایشی داده‌های عمومی ابداع شد و عمدتاً برای توصیف مجموعه داده‌های[2] زیاد استفاده می‌شود. در این فصل، تعاریف کلان داده را معرفی می‌کنیم و تکامل آن‌را در ۲۰ سال اخیر مرور می‌کنیم. به طور خاص، ویژگی‌های کلان داده، همچنین ویژگی‌های 4V آن شامل حجم[3]، تنوع، سرعت و ارزش را معرفی می‌کنیم. چالش‌های مربوط با کلان داده نیز در این فصل معرفی می شوند.

۱-۱- طلوع دوران کلان داده

در ۲۰ سال اخیر، داده‌ها در مقیاس بزرگی در رشته‌های مختلف افزایش یافته‌اند. براساس یک گزارش از شرکت داده‌های بین المللی (IDC[4]) در سال ۲۰۱۱ کل داده‌های تولید شده و کپی شده در دنیا 1.8ZB حدود 10^{21}B می‌باشد که در طول ۵ سال ۲ برابر شده است. این رقم در دو سال آینده دو برابر خواهد شد.

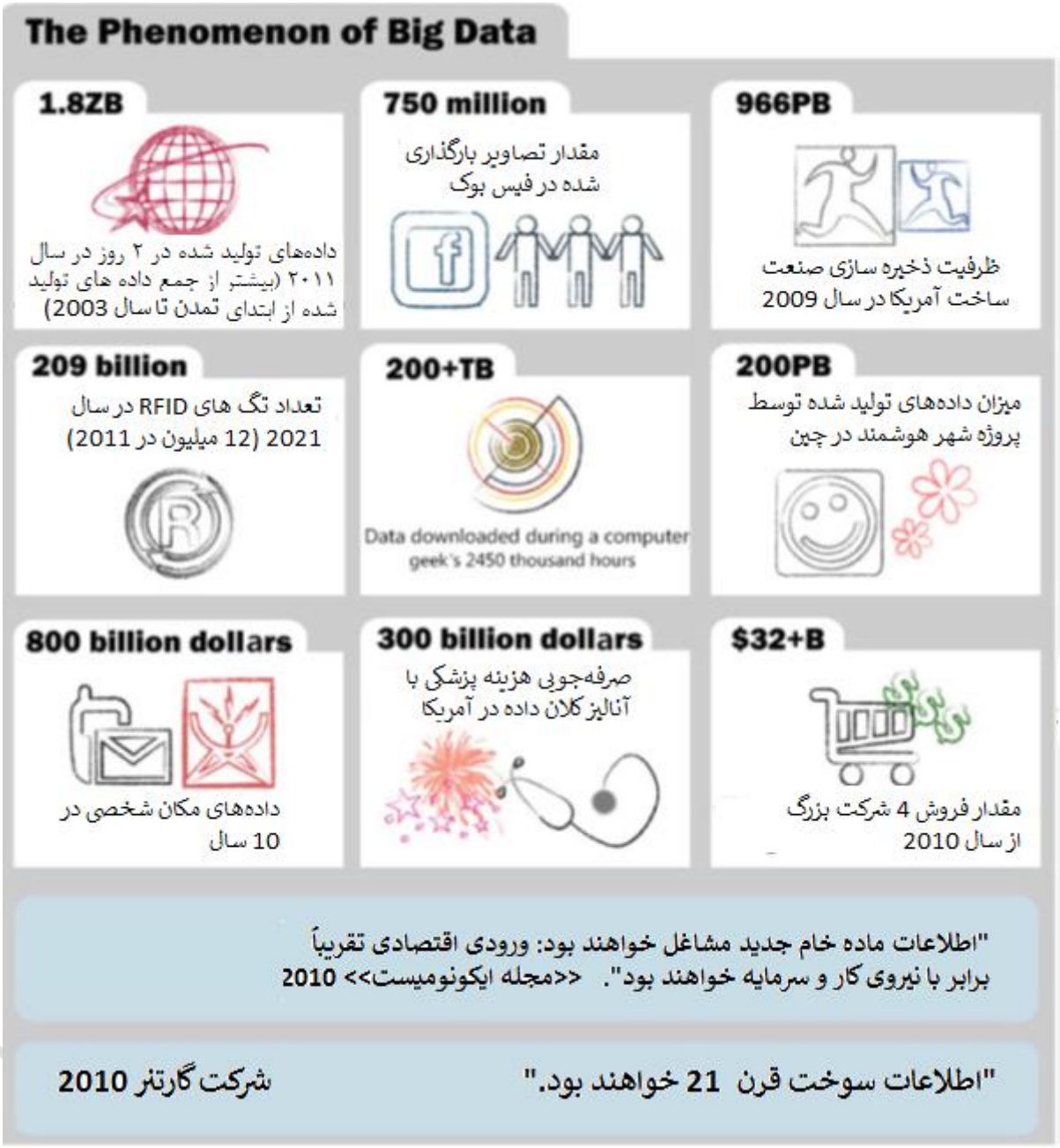
اصطلاح کلان داده به خاطر انفجار افزایشی داده‌های عمومی ابداع شد و عمدتاً برای توصیف مجموعه داده‌های زیاد استفاده می‌شود. در مقایسه با مجموعه داده‌های سنتی، کلان داده عموماً نیازمند انبوه داده‌های ساختارنیافته می‌باشد که نیازمند تجزیه تحلیل بلادرنگ[5] بیشتری می‌باشد. مجلات زیادی مثل Economist, New York Time و غیره راجع به کلان داده مطلب می‌نویسند و در ژورنال‌های علمی معتبر مطلب در مورد آن چاپ می‌شود. بسیاری از آژانس‌های دولتی نقشه‌هایی برای شتاب در تحقیقات و کاربردهای کلان داده دارند و صنایع نیز در مورد پتانسیل کلان داده علاقه مند هستند.

رشد حجم داده‌های تولید شده بسیار زیاد می‌باشد و این چالش‌هایی در پی دارد و متقاضی راه‌حل‌های سریع است:

۱- توسعه‌های اخیر در فنآوری IT به راحتی داده تولید می‌کنند، بطور مثال در هر دقیقه ۷۲ ساعت ویدیو در یوتیوب بارگذاری می‌شود که با چالش جمع‌آوری و تجمع حجم عظیمی از داده‌ها از منابع داده توزیع شده مواجه هستیم.
۲- داده‌های جمع‌آوری شده بطور افزایشی در حال زیاد شدن هستند که باعث مساله چگونگی ذخیره و مدیریت مجموعه داده نامتجانس حجیم هستیم که نیازمند نیازها و زیرساخت‌های سخت افزاری و نرم‌افزاری می‌باشد.
۳- با در نظر گرفتن عدم تجانس، مقیاس‌پذیری، بلادرنگ بودن، پیچیدگی، و محرمانگی چنین داده‌های کلانی نیاز به داده‌کاوی موثر در سطوح مختلف تجزیه تحلیل، مدل‌سازی، پیش‌بینی، و روش‌های بهینه سازی دارند تا بتوانیم تصمیم گیری را بهبود دهیم.

[1] Big Data
[2] Dataset
[3] Volume-Variety- Velocity, and Value= 4V
[4] International Data Corporation
[5] Real- Time

رشد سریع رایانش ابری و اینترنت اشیاء، IOT[1] باعث تسریع رشد داده‌ها می‌شود. رایانش ابری، محافظت امن، سایت‌های دسترسی، و کانال‌ها را برای مجموعه داده‌ها فراهم می‌کند. در IOT سنسورها در همه جهان در حال جمع‌آوری و ارسال داده‌ها هستند که باید در ابر پردازش و ذخیره شوند، شکل ۱-۱ توسعه حجم داده‌های عمومی را نشان می‌دهد.



شکل ۱-۱ نمایش افزایش مداوم داده‌ها

[1] Internet of things

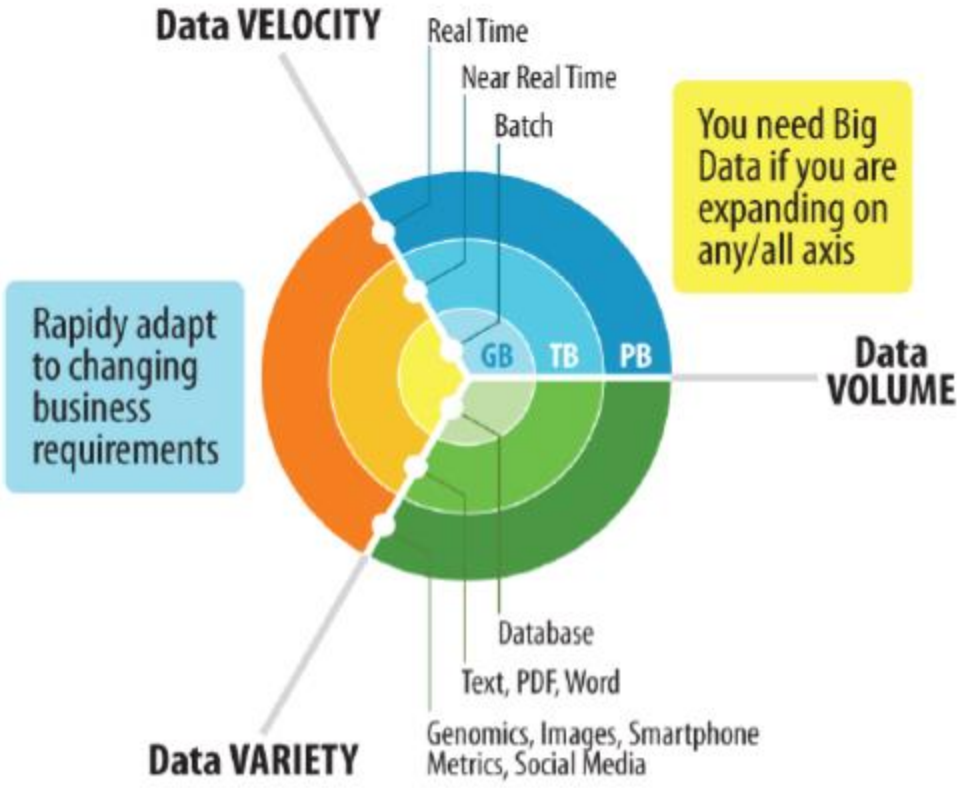
۱-۲ تعریف و ویژگی‌های کلان داده

کلان داده یک مفهوم انتزاعی است و تعاریف مختلفی برای آن مطرح شده است مهمترین آن این است: ***داده‌هایی هستند که نمی‌توانند به روش‌های IT سنتی مشاهده، جمع‌آوری، مدیریت و پردازش شوند***. از دید حجم داریم:

۱- حجم‌های مجموعه داده که با استاندارد کلان داده تطابق دارد در حال تغییر هستند و ممکن است در زمان و با توسعه فن‌آوری تغییر کنند.

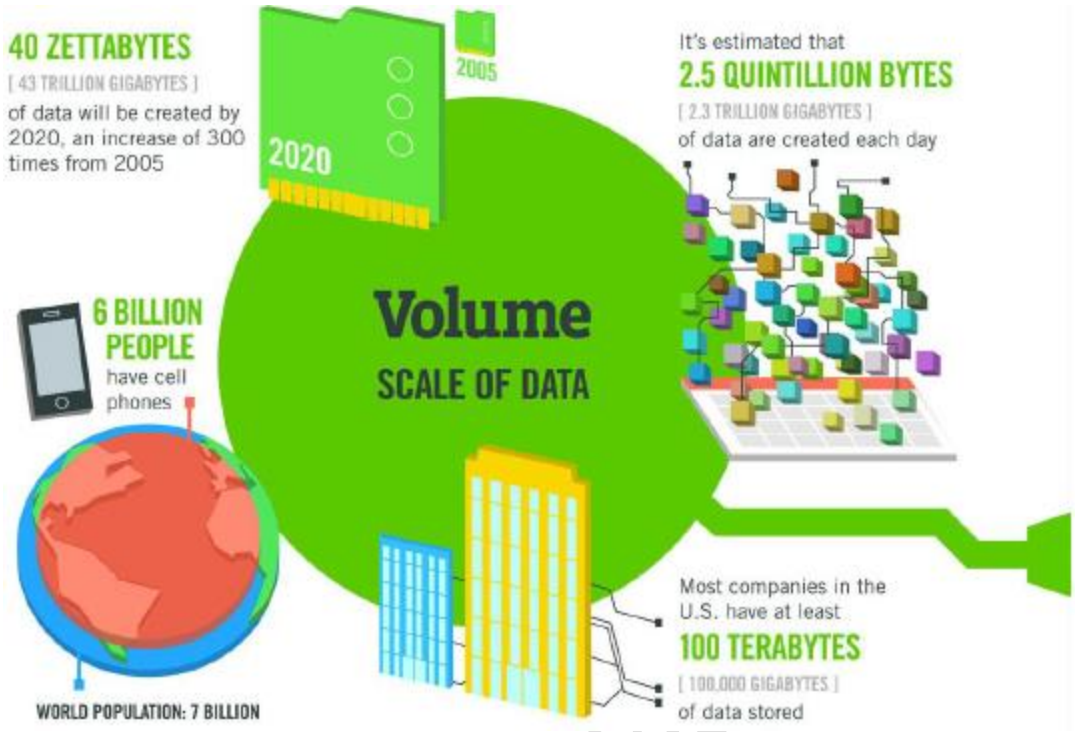
۲- حجم‌های مجموعه داده که با استاندارد کلان داده تطابق دارند ممکن است در کاربردهای مختلف تغییر کنند.

برای کلان داده مدل‌های مختلفی ارائه شده است مثل مدل 3V که شامل Volume: حجم رو به افزایش است، Velocity: سرعت پردازش باید سریع باشد، Variety: تنوع مشخص کننده انواع مختلف داده‌ها است که شامل ساختار یافته سنتی، نیمه ساختار یافته و ساختارنیافته مثل صدا، تصویر، ویدیو، صفحه وب، متن می‌باشد. این مدل در شکل ۱-۲ نشان داده شده است:



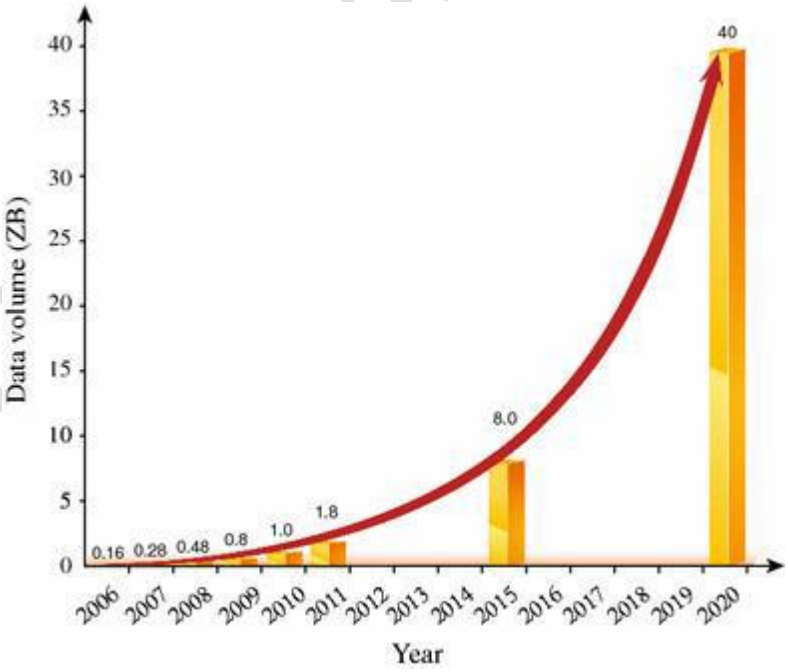
شکل ۱-۲ مدل 3V برای کلان داده.

در شکل ۱-۳ تخمین حجم داده‌های تولیدی در سال ۲۰۲۰ نشان داده شده است:



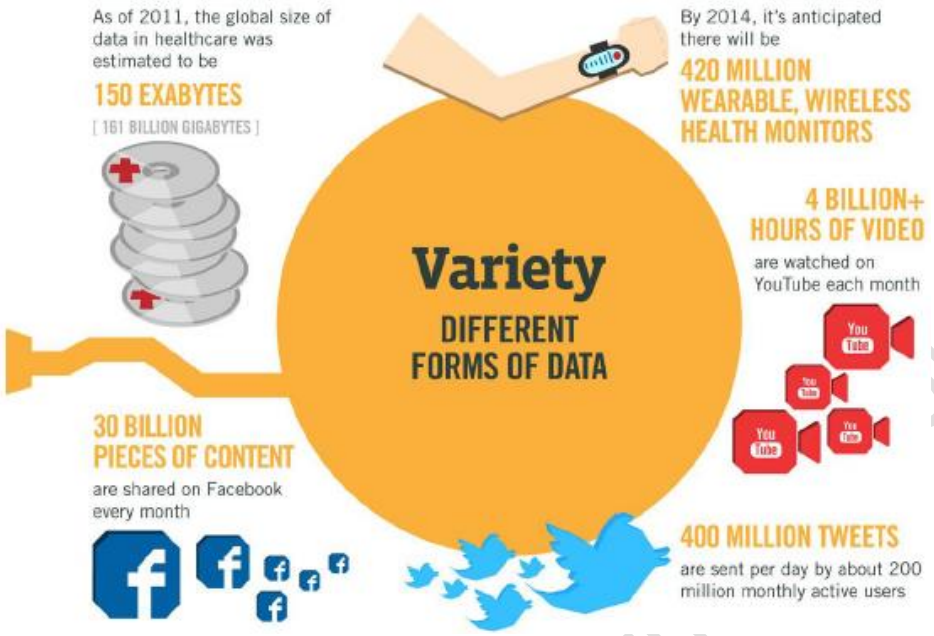
شکل ۱-۳ حجم داده‌های تخمینی در سال ۲۰۲۰

در نمودار ۱-۱ روند رشد کلان داده نشان داده شده است.



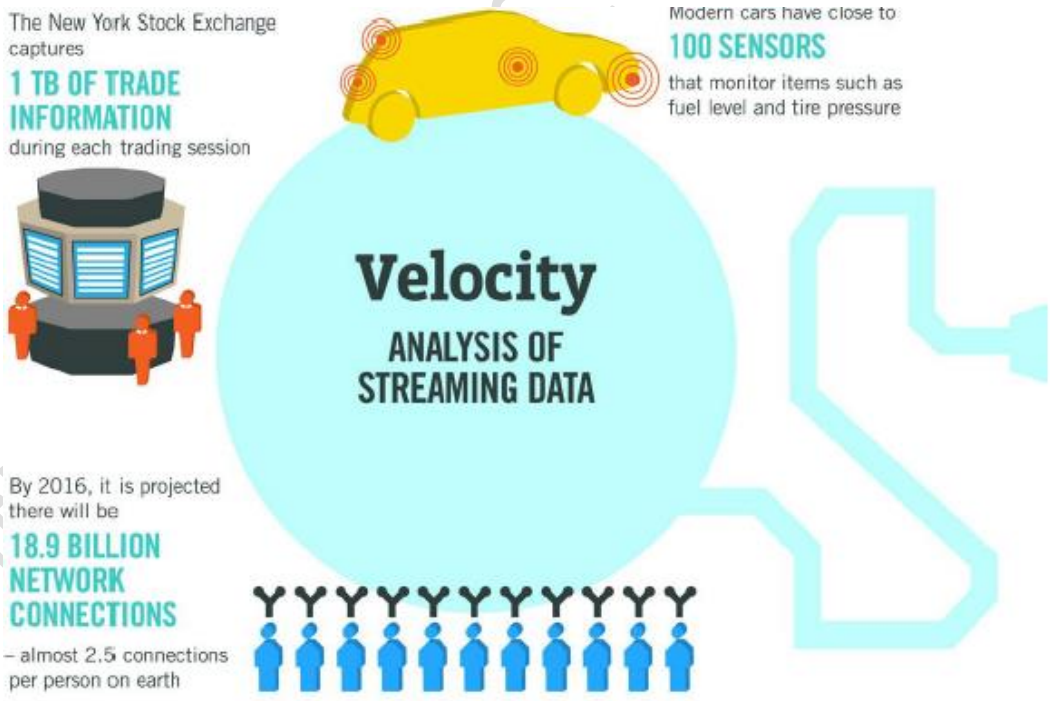
نمودار ۱-۱ روند رشد حجم تخمینی کلان داده

در شکل ۱-۴ تنوع داده های استفاده شده در اینترنت نشان داده شده است.



شکل ۱-۴ تنوع داده های استفاده شده در اینترنت در رشته های مختلف

در شکل ۱-۵ سرعت تبادل داده‌ها در حوزه های مختلف نشان داده شده است. در بعضی رشته ها مثل کنفرانس ویدئویی و یا تراکنش های مالی نیازمند پردازش سریع حجم انبوهی از داده‌ها هستیم.

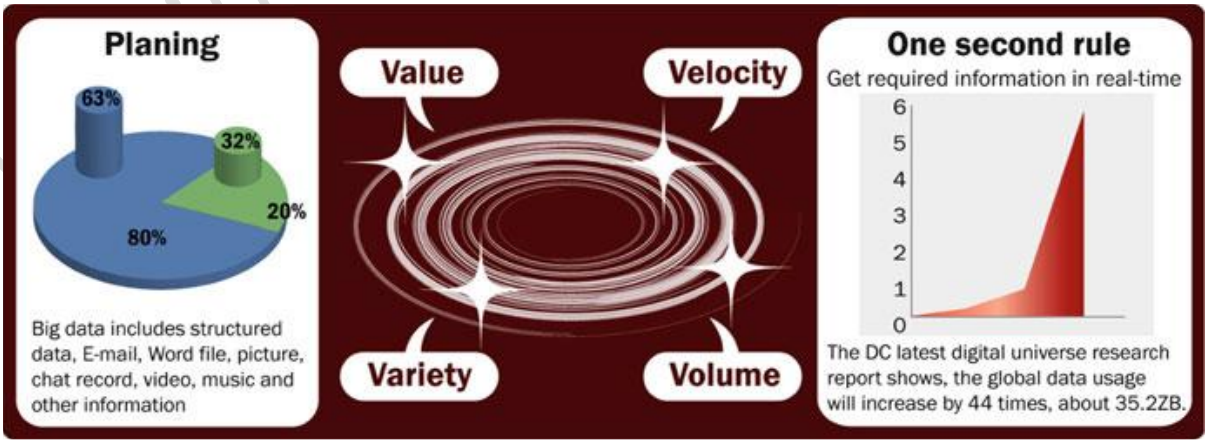


شکل ۱-۵ سرعت تبادل داده‌های استفاده شده درحوزه های مختلف.

در شکل ۱-۶ آنچه در یک دقیقه در اینترنت در سال ۲۰۱۷ اتفاق افتاده است نشان داده شده است:

شکل ۱-۶ آنچه در یک دقیقه در اینترنت در سال ۲۰۱۷ اتفاق افتاده است.

برای مدل سازی کلان داده از روش های دیگری نیز استفاده می‌شود که هر مدل وابسته به تعریفی است که از کلان داده ارائه شده است به عنوان مثال تعریفی که IDC برای کلان داده ارائه داده است عبارت است از: ***فن آوری کلان داده نسل جدیدی از فن‌آوری و معماری‌ها را معرفی می‌کند و طوری طراحی شده است تا بطور مقتصدانه ارزش را از حجم زیاد و متنوع داده با تشخیص با سرعت بالا و/یا تجزیه تحلیل استخراج کند***. در این تعریف کلان داده بصورت مدل 4V شامل Volume:حجم، Variety: تنوع، Velocity: سرعت تولید و Value: ارزش تعریف می‌شود. این تعریف بحرانی‌ترین مساله را در کلان داده مشخص می‌کند: یعنی بدست آوردن ارزش از مجموعه داده‌ها در مقیاس فوق العاده زیاد با انواع متفاوت و سرعت تولید بالا. شکل ۱-۷ این مدل را نشان می‌دهد.



شکل ۱-۷: مدل 4V از ویژگی‌های کلان داده.

۱-۳ کاربردهای کلان داده: با توجه به مطالب گفته شده مشاهده می شود که حوزه کلان داده بسیار وسیع و دامنه کاربرد آن گسترده است. نکته دیگر افزایش روز افزون حوزه های دیگر به سمت کلان داده است. از جمله حوزه های کلیدی که از کلان داده استفاده می کنند می‌توان به حوزه بانکداری- پزشکی و سلامت، رسانه اجتماعی، اینترنت اشیاء، شبکه های اجتماعی و ... اشاره کرد. این حوزه ها بطور خلاصه در شکل ۱-۸ نشان داده شده‌اند.

شکل ۱-۸ حوزه های مختلفی که از کلان داده استفاده می‌کنند.

۱-۳ چالش‌های کلان داده

با توجه به تعاریف مطرح شده در حوزه کلان داده و همچنین گستردگی داده های تولید شده چالش هایی در این حوزه وجود دارند که باید روش‌هایی برای رفع و رجوع آنها ارائه شود. این چالش‌ها در هنگام گردآوری داده‌ها[1]، ذخیره‌سازی[2]، مدیریت[3] و تجزیه تحلیل[4] داده‌ها ایجاد می‌شوند. چالش‌هایی که حوزه کلان داده با آن مواجه می‌باشد عبارتند از:

- **نمایش داده**[5]: بسیاری از مجموعه داده‌ها شامل سطوح خاصی از عدم تجانس در نوع، ساختار، معنی، سازمان‌دهی، دانه-بندی، و قابلیت دسترس‌پذیری می‌باشند. هدف نمایش داده این است که داده‌ها برای تجزیه تحلیل کامپیوتری و تفسیر کاربر با معنی‌تر به نظر برسند. به علاوه، نمایش نامناسب داده‌ها، ارزش داده‌ها را کاهش می‌دهد و ممکن است مانع تجزیه تحلیل موثر داده‌ها شود. نمایش موثر داده‌ها ساختار داده‌ها، کلاس، نوع آنها و تجمع فن‌آوری‌ها را انعکاس خواهد داد.
- **کاهش افزونگی و فشرده‌سازی داده‌ها**[6]: معمولاً سطح قابل توجهی از افزونگی در مجموعه داده‌ها وجود دارند. کاهش افزونگی و فشرده‌سازی داده‌ها برای کاهش هزینه غیرمستقیم کل سیستم موثر است بالاخص اگر ارزش داده‌ها تحت تاثیر

[1] data acquisition
[2] Storage
[3] Management
[4] Analysis
[5] Data Representation
[6] Redundancy Reduction and Data compression

قرار نگیرد. به عنوان مثال اکثر داده‌های تولید شده توسط شبکه سنسورها خیلی افزونه هستند که ممکن است در یک سطحی فشرده سازی و فیلتر شوند.

- **مدیریت چرخه عمر داده‌ها**[1]: در مقایسه با پیشرفته‌های نسبتاً کُند در سیستم های ذخیره‌سازی، سنسورهای فراگیر شده و محاسبات، داده‌ها را در مقیاس و سرعت بالا تولید می‌کنند. ما با چالش‌های زیادی مواجه هستیم که یکی از آنها این است که سیستم ذخیره سازی موجود نمی‌تواند چنین حجمی از داده‌ها را پشتیبانی کند. بطور کلی ارزش مخفی در کلان داده وابسته به تازگی داده‌ها می‌باشد. بنابراین مفاهیم مهمی مرتبط با ارزش روش‌های تجزیه تحلیل[2] باید ابداع شود تا تصمیم بگیرد چه داده‌هایی باید ذخیره شوند و چه داده‌هایی باید حذف شوند.
- **مکانیزم‌های روش تجزیه تحلیل:** سیستم‌های تحلیلی کلان داده باید انبوهی از داده‌های نامتجانس را در زمانی محدود پردازش کنند. به هر حال RDBMSهای سنتی با کمبود مقیاس‌پذیری و قابلیت توسعه طراحی شده‌اند که نمی‌توانند نیازهای کارایی را برآورده کنند. پایگاه داده‌های غیر رابطه‌ای در تجزیه تحلیل داده‌های غیرساختار یافته مزایایی دارند و در کلان داده‌ها استفاده می‌شوند. البته این پایگاه داده‌ها کارایی پایگاه داده‌های رابطه ای را ندارند و بعضی سرمایه گذاران از پایگاه داده‌های ترکیبی استفاده می‌کنند تا مزایای هر دو را استفاده کنند (مثل فیس بوک و تابو). تحقیقات بیشتری برای پایگاه داده در-حافظه [3]و داده‌ها براساس تجزیه تحلیل تقریبی مورد نیاز است.
- **محرمانگی داده‌ها**[4]: بسیاری از تهیه کنندگان سرویس کلان داده‌ها به خاطر ظرفیت محدود شده آنها، در حال حاضر نمی‌توانند بطور موثری چنین حجم انبوهی از داده‌ها را نگهداری و تجزیه تحلیل کنند. آنها باید به افراد خبره یا ابزارهایی تکیه کنند تا داده‌ها را تجزیه تحلیل کنند، که این ریسک‌های امنیتی را افزایش می‌دهد. بنابراین باید واحدهای خوبی برای تضمین امنیت داده‌ها ایجاد شوند.
- **مدیریت انرژی:** انرژی مصرفی سیستم‌های محاسباتی مین فریم‌ها هم از دید اقتصادی و هم محیطی توجه زیادی را به خود جلب کرده است. با افزایش حجم داده‌ها و نیازهای تجزیه تحلیلی، پردازش، ذخیره سازی، و انتقال کلان داده مصرف توان و مدیریت برق بیشتری نیاز دارند. بنابراین باید مکانیزم‌های کنترل مصرف توان سطح-سیستم و مدیریتی برای کلان داده‌ها برقرار شوند در حالیکه قابلیت توسعه و قابل دسترس بودن هر دو تضمین شود.
- **قابلیت توسعه و مقیاس‌پذیری**[5]: سیستم‌های تجزیه تحلیلی کلان داده‌ها باید مجموعه داده‌های حال و آینده را پشتیبانی کنند. الگوریتم‌های روش تجزیه تحلیل باید قادر باشند مجموعه داده‌های پیچیده‌تر و رو به توسعه‌تر را پردازش کنند.
- **همکاری**[6]: تجزیه تحلیل کلان داده‌ها یک حوزه تحقیقاتی فراگیر است که نیازمند این است که خبرگان در رشته‌های مختلف با هم کار کنند تا پتانسیل کلان داده را به خوبی استفاده کنند. یک معماری شبکه کلان داده جامع باید ایجاد

[1] Data Life Cycle Management
[2] Analytic
[3] In- memory
[4] Data Confidentiality
[5] Expendability and Scalability
[6] Cooperation

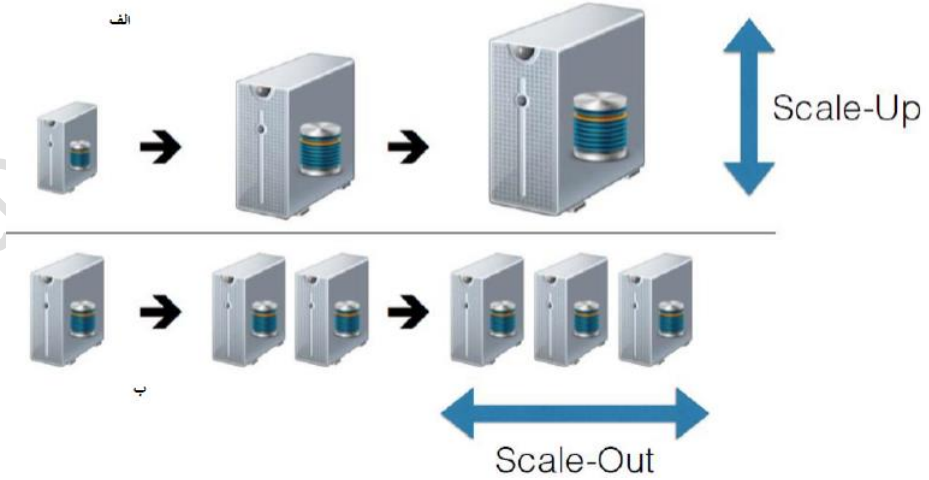
شود تا به مهندسین و دانشمندان در رشته‌های مختلف کمک کند تا به انواع مختلف داده‌ها دسترسی داشته باشند و سرمایه‌شان را به خوبی بهره برداری کنند.

فصل دوم: روش‌های ذخیره سازی و مدیریت کلان داده

ذخیره سازی داده‌ها اشاره به ذخیره‌سازی و مدیریت مجموعه داده‌های مقیاس بالا دارد در حالیکه دارای قابلیت اطمینان و دسترس‌پذیری است. زیرساخت سخت افزاری شامل تکنولوژی منابع ارتباط اطلاعات مشترک انبوه (ICT) است که برای بازخورد نیازهای فوری کارها بهره وری می‌شود، و چنین منابع ICT به روشی کشسان[1] سازماندهی می‌شوند. زیرساخت سخت افزاری باید کشسانی و قابل پیکربندی مجدد پویا را داشته باشد تا با محیط‌های کاربردی مختلف تطابق داشته باشد. روش‌های ذخیره‌سازی داده در بالای زیرساخت سخت افزاری استفاده می‌شوند تا مجموعه داده‌های در مقیاس بزرگ را نگه‌داری کنند. سیستم-های ذخیره سازی باید با رابط‌های زیادی تجهیز شوند تا پرس‌و جوها سریع باشند.

کلان داده با رشد انفجاری داده‌ها مواجه است. داده‌ها با توجه به رشد پرشتاب نیازهای ضروری روی ذخیره سازی و مدیریت دارند. بطور رایج، تجهیزات ذخیره داده‌ها فقط تجهیزات جانبی سرورها هستند و اکثر آنها داده‌های ساختار یافته RDBMS را ذخیره، مدیریت، جستجو و تجزیه تحلیل می‌کنند. با توجه به رشد سریع کلان داده به GB, TB,PB تجهیزات ذخیره سازی سنتی برای مدیریت آنها کافی نیستند. اهمیت تجهیزات ذخیره سازی روز افزون است و هزینه ذخیره سازی مهمترین چالش بسیاری از شرکت-های اینترنتی شده است. بنابراین تحقیق رو مخزن داده‌ها ضروری است.

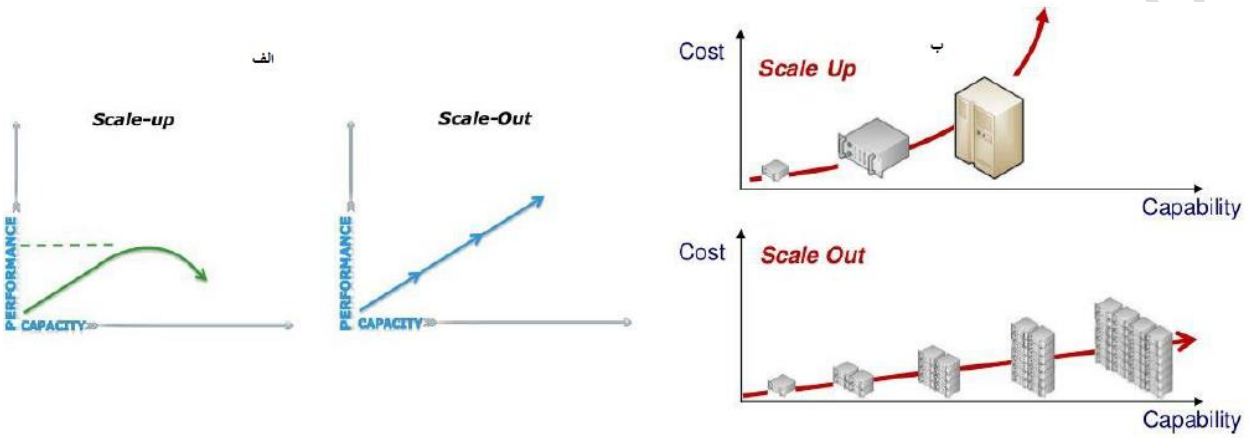
با توجه به چالش‌های مطرح شده برای حوزه کلان داده باید روش‌هایی برای حل و فصل آنها ایجاد کنیم. یکی از مسائل در این حوزه ذخیره سازی داده‌ها می‌باشد. این حجم عظیم داده‌ها را در کجا ذخیره کنیم؟ یکی از روش‌هایی که به ذهن می‌رسد افزایش حجم رسانه‌ی ذخیره سازی[2] (دیسک‌ها) می‌باشد. در این روش که در شکل ۲-۱-الف نشان داده شده است سیستم دیسک بزرگتر و قاعدتاً گران‌تر خواهد شد. این باعث مدیریت متمرکز روی دیسک می‌شود اما لزوماً باعث افزایش سرعت نخواهد شد چرا که خود دیسک یک نقطه شکست برای ما ایجاد خواهد کرد که در صورت خرابی سیستم از کار خواهد افتاد. روش دیگر *افزایش تعداد دیسک‌ها بصورت توزیع شده*[3] می‌باشد که در شکل ۱-۲ب نشان داده شده است. در این روش مدیریت مشکل تر خواهد بود ولی با مدیریت صحیح می‌توان از قابلیت موازات آنها جهت افزایش سرعت استفاده کرد همچنین با کپی کردن داده‌ها در دیسک‌های مختلف می‌توان از حذف داده‌ها در صورت خرابی یک دیسک یا گروهی از دیسک‌ها جلوگیری کرد.



شکل ۲-۱- در شکل الف حجم دیسک افزایش می‌یابد و در شکل ب تعداد دیسک ها زیاد می‌شود.

[1] Elasticity
[2] Scale-Up
[3] Scale-Out

در شکل ۲-۲ بین Scale-out و Scale-Up مقایسه داریم. همانطور که در شکل ۲-۲-الف مشاهده می‌کنید با افزایش ظرفیت کارایی هر دو افزایش می‌یابد اما در Scale-Up بعد از یک حد آستانه، کارایی افت خواهد داشت. علت آن متمرکز بودن مدیریت و عدم پاسخ به کارهای موازی می‌باشد. اما در Scale-Out کارایی بطور خطی با افزایش حجم زیاد می‌شود. در شکل ۲-۳-ب این دو از لحاظ قیمت مقایسه شده‌اند. آیا خریدن یک دیسک بسیار حجیم بهتر است یا خریدن چندین دیسک ارزان؟ در مورد Scal-Up قیمت بصورت شبه نمایی رشد می‌کند اما در Scale-Out قیمت بصورت خطی رشد می‌کند. بنابراین بهتر هست که سیستم بصورت Scale-Out طراحی شود تا هم بتوانیم از قابلیت موازات آن استفاده کنیم هم بتوانیم از حجم بیشتری بهره ببریم و هم هزینه خیلی زیادی پرداخت نکنیم.



شکل ۲-۲ مقایسه بین Scale-Out و Scale-Up

پردازش کلان داده‌ها:

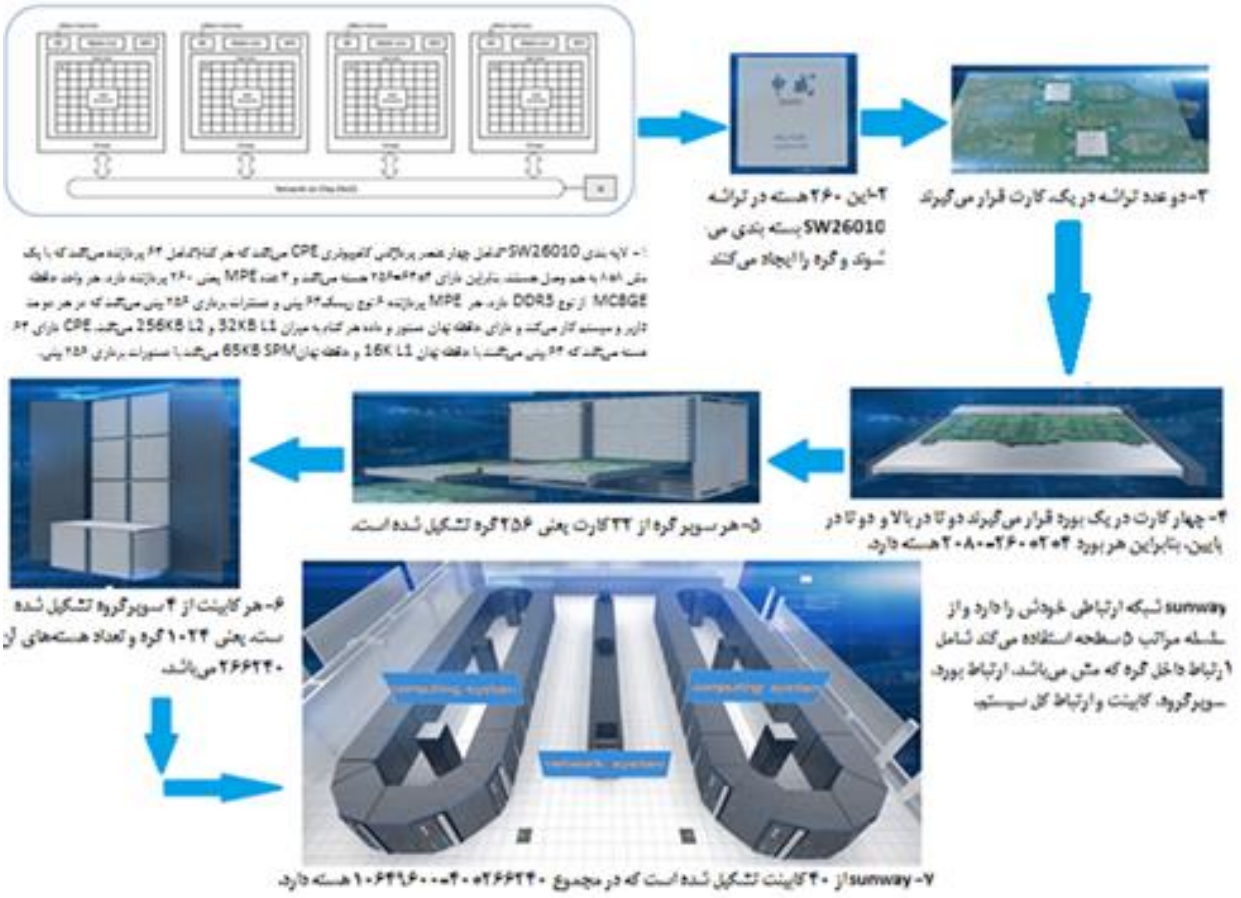
برای پردازش کلان داده‌ها هم مثل ذخیره سازی می‌توان عمل کرد آیا تعداد زیادی پردازنده به صورت توزیع شده در سیستم‌های مختلف وظیفه پردازش را برعهده داشته باشند یا یک سیستم بسیار قوی از پردازنده‌ها این وظیفه را برعهده داشته باشد؟ در مورد دوم معمولاً از سوپر کامپیوترها استفاده می‌شود که سیستمی بسیار قوی و سریع از لحاظ محاسباتی است و دارای قیمت بالایی می‌باشد. در جدول ۲-۱ سرعت ۵ سوپر کامپیوتر اول دنیا را با هم مقایسه کرده‌ایم:

سوپرکامپیوتر /خصوصیت	Sunway Taihulight	Tianhe-2 (MikyWay-2)	Titan-Cray XK7	Sequoia-BlueGene/Q,	Cori-Cray XC40
سازنده	NRCPC	NUDT	Cray شرکت	IBM	IBM
پردازنده	Sunway SW26010 260C 1.45Ghz	Intel Xeon E5 2692 12C 2.200GHz	Opteron 6274 16c 2.200GHz	Power BQC 16C 1.60GHz	Intel Xeon Phi 7250 68C 1.4GHz
تعداد پردازنده	10,649,600	3,120,000	560640	1,572,864	622,336
کارایی (Rmax)	93,014.6TFlops	33,862.7TFlop/s	17,590TFlop/s	17,173.2TFlop/s	14,014.7TFlop/s
حداکثر تئوری	125,435.9Tflops	54,902.7TFlop/s	27,112.5TFlop/s	20,132.7TFlop/s	27,880.7TFlop/s
توان	15,371kW	17,808.00kW	8,209.00 kW	7,890.00 kW	3,939.00Kw
حافظه	1,310,720 GB	1,024,000GB	710,144 GB	1,572,864 GB	878,592GB
شبکه اتصالی	Sunway	TH Express-2	Cray Gemini	Custom	Aries interconnect
سیستم عامل	Sunway RaiseOS 2.0.5	Kylin Linux	Cray Linux	Linux	Cray Linux
کشور	china	Changsha چین شهر	United States شهر Oak Ridge	United States شهر livemore	Unites states شهر Argonne

جدول ۲-۱ مشخصات پنج تا از بهترین سوپرکامپیوترهای دنیا (راست به چپ) در نوامبر ۲۰۱۶ برگرفته از سایت http://www.top500.org

همانطور که مشاهده می شود بهترین سوپر کامپیوتر دنیا در حال حاضر Sunway چین می‌باشد با ۱۰۶۹۴۶۰۰ هسته پردازنده با قدرت پردازش 93014TFlops. برای مقایسه سرعت سوپرکامپیوتر اول دنیا در تاریخ فوق الذکر با یک PC که سرعت آن 1GFlops می‌باشد متوجه می‌شویم کاری را که این سوپر کامپیوتر در یک ثانیه انجام می‌دهد PC آن‌را در ۳ سال انجام خواهد داد. اما قیمت این سوپر کامپیوتر ۳۰۰میلیون دلار است. آیا از یک سوپر کامپیوتر ۳۰۰ میلیون دلاری استفاده کنیم یا ۳۰۰ هزار سیستم ارزان هزار دلاری؟

سوپر کامپیوترها چگونه ساخته می شوند در شکل ۲-۳ روند ساخت سوپر کامپیوتر sunway را نشان داده ایم. توان مصرفی و هزینه های خنک سازی یکی از مهمترین چالش های پیش روی یک سوپر کامپیوتر است.

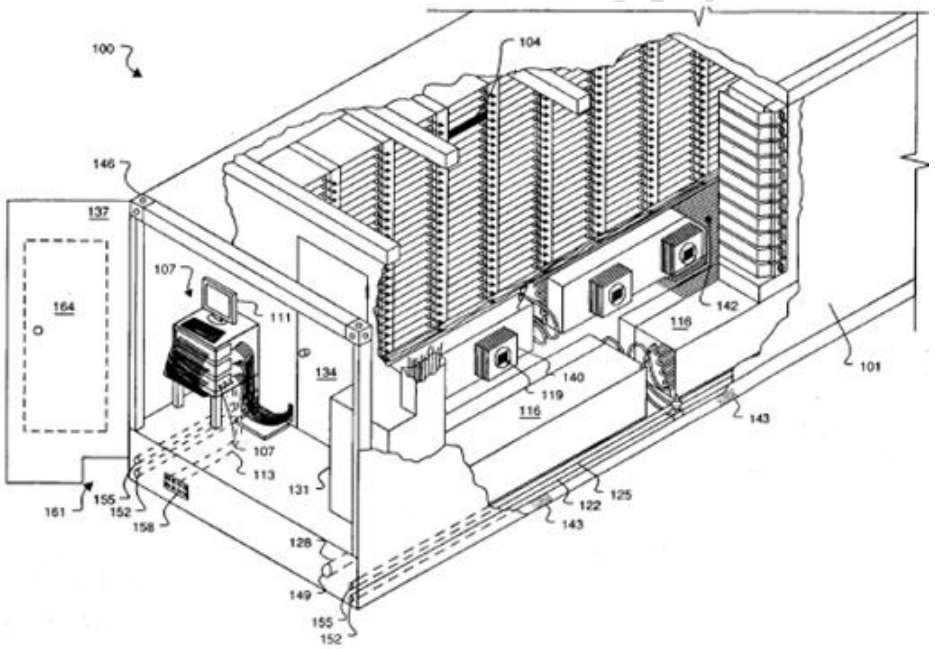


شکل ۲-۳ روند ساخت سوپرکامپیوتر sunway

شکل ۲-۴ نمونه یک سوپر کامپیوتر

اما سوپرکامپیوتر برای کارهای پردازشی کلان داده مناسب خیلی نیست. چرا که بسیار گران قیمت می‌باشد. بهتر هست از تعدادی کامپیوتر ارزان قیمت در یک سوله (warehouse) استفاده شود. به عنوان مثال گوگل و میکروسافت WSCهایی با استفاده از کانتینرهای کشتیرانی ساخته‌اند. ایده‌ی ساخت یک WSC از کانتینرها ماژولی ساختن WSC[1] می‌باشد. هر کانتینر مستقل است، و تنها ارتباطات بیرونی، شبکه، توان و آب می‌باشد. کانتینرها، شبکه، توان و خنک سازی را برای سرورهای داخلی‌شان آماده می‌کنند، بنابراین وظیفه‌ی یک WSC فراهم کردن شبکه، توان و آب سرد به کانتینر و پمپ آب گرم منتج به برج‌ها و چیلرهای خنک کننده بیرونی است.

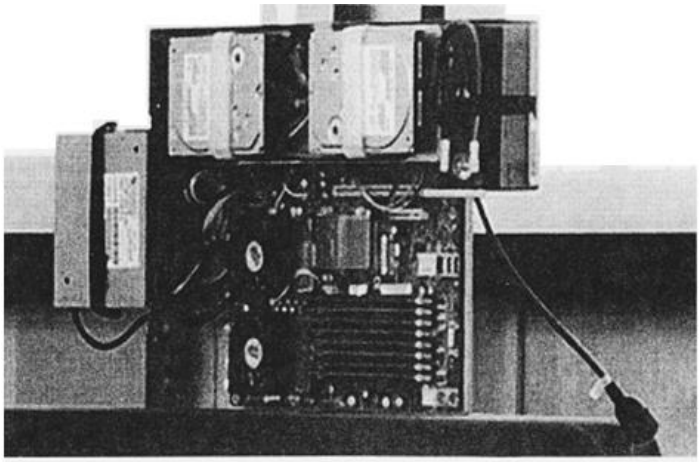
WSCی گوگل که ما آن‌را مرور می‌کنیم شامل ۴۵ کانتینر با طول ۴۰–فوت در یک فضای ۳۰۰ در ۲۵۰ فوتی یا ۷۵۰۰۰فوت مربع (حدوداً ۷۰۰۰ متر مربع) می‌باشد. در یک ورهاوس (سوله) ، ۳۰ کانتینر قرار دارد که هر دو تا با هم بصورت پشته شده قرار گرفته‌اند، یعنی ۱۵ زوج کانتینر روی هم قرار دارند. گوگل مکان WSC را فاش نکرد، اما زمانی ساخته شد که گوگل WSCها را در Dalles، Oregon توسعه داد که آب و هوای معتدل دارند و نزدیک منبع هیدروالکتریک و فیبر ستون فقرات اینترنت است. شکل ۲–۵ یک نمای نیم‌رخ از کانتینر گوگل را نشان می‌دهد. یک کانتینر حداکثر ۱۱۶۰ سرور را نگه می‌دارد، بنابراین کانتینرها برای ۵۲۰۰۰ سرور فضا دارند. (این WSC حدوداً ۴۰۰۰۰ سرور دارد). سرورها بصورت ۲۰ تایی در رک‌ها پشته شده‌اند و در دو ردیف دراز که هر کدام ۲۹ رک (که به آن قفسه نیز می‌گویند) دارند شکل دهی می‌شوند. سوییچ‌های رک ۴۸–پورته، سوییچ‌های اترنت ۱گیگا بیت/ثانیه هستند، که در هر رک قرار می‌گیرند.



شکل ۲–۵: گوگل استاندارد 1AAA را برای کانتینرها سفارشی کرده است: (40×8×9.5 feet (12.2×2.4×2.9 meters. سرورها بصورت ۲۰تایی در رک‌ها پشته شده‌اند. دو ردیف دراز از رک‌ها داریم. در هر ردیف ۲۹ رک وجود دارد، هر ردیف در یک طرف کانتینر قرار دارد. یک راه‌رو خنک ساز در وسط کانتینر وجود دارد که هوای گرم را به خارج می‌برد. ساختار آویزان رک تعمیر سیستم خنک سازی را بدون برداشتن سرورها ساده‌تر می‌کند. برای اینکه به افراد داخل کانتینر اجازه تعمیر اجزاء داده شود، کانتینر شامل سیستم‌های امنیتی برای تشخیص آتش و حذف غبار، خروجی و نور اضطراری و بستن توان اضطراری می‌باشد

سرورهای استفاده شده در این ورهاوس سیستم های ساده می‌باشند . شکل ۲–۶ سرور طراحی شده توسط گوگل برای این WSC را نشان می‌دهد. برای بهبود بازدهی، منبع تغذیه فقط ۱۲ ولت برای مادربورد تهیه می‌کند و مادربورد ولتاژ کافی برای دیسک‌هایی که روی بورد دارد تهیه می‌کند.

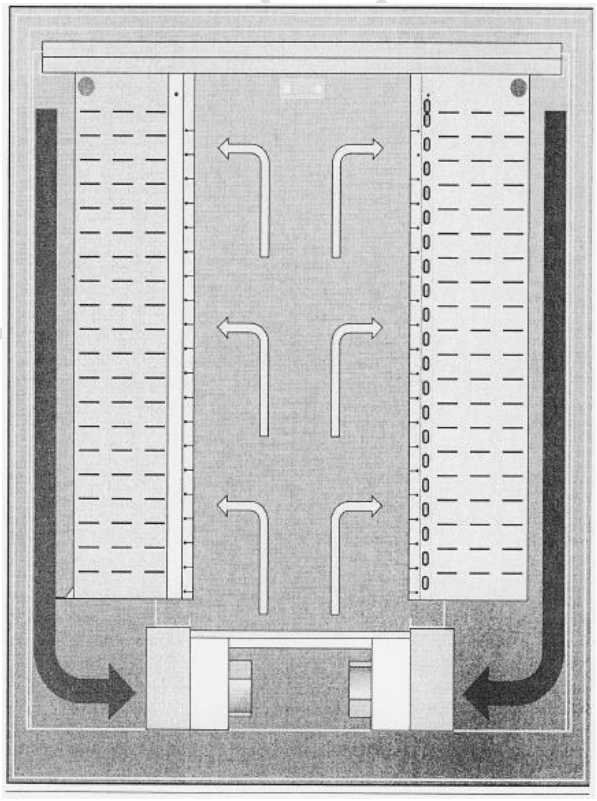
[1] Warehouse scale computers



شکل ۲-۶: سرورِ WSCای گوگل. منبع تغذیه در سمت چپ است و دو دیسک در بالا هستند. دو فن زیر دیسک چپ دو سوکت ریزپردازنده AMD Barcelona که هر کدام با دو هسته که با سرعت 2.2Ghz کار می‌کنند را پوشش می‌دهند. هشت DIMM در سمت راست پایین که هر کدام 1GB نگه می‌دارند، ظرفیت کلی 8GB دارند. هیچ ورقه اضافی وجود ندارد، سرورها به داخل باتری وصل می‌شوند و یک پلنیوم مجزا در رک برای هر سرور وجود دارد که به کنترل جریان هوا کمک می‌کند. به خاطر ارتفاع باتری‌ها، ۲۰ سرور در یک رک قرار می‌گیرد.

خنک سازی و توان در WSCای گوگل

شکل ۲-۷ یک برش مقطعی از کانتینر است که جریان هوا را نشان می‌دهد. رک‌های کامپیوتر به سقف کانتینر متصل شده‌اند. در وسط کانتینر،کف برآمده است و خنک کننده زیر کف قرار دارد و هوا را به داخل راه‌روی بین رک‌ها می‌دمد. هوای گرم از پشت رک‌ها برگشت داده می‌شود. فضای محدود شده کانتینر از ترکیب هوای سرد و گرم جلوگیری می‌کند که بازدهی خنک سازی را بهبود می‌دهد. فن‌های با سرعت متغیر با کمترین سرعتی که خنک کردن رک نیازمند است کار می‌کنند نه با سرعت ثابت. اگر هوای بیرون خیلی سرد باشد، برج‌های خنک کننده نیازمند بخاری‌هایی برای اجتناب از یخ زدن می‌باشند.



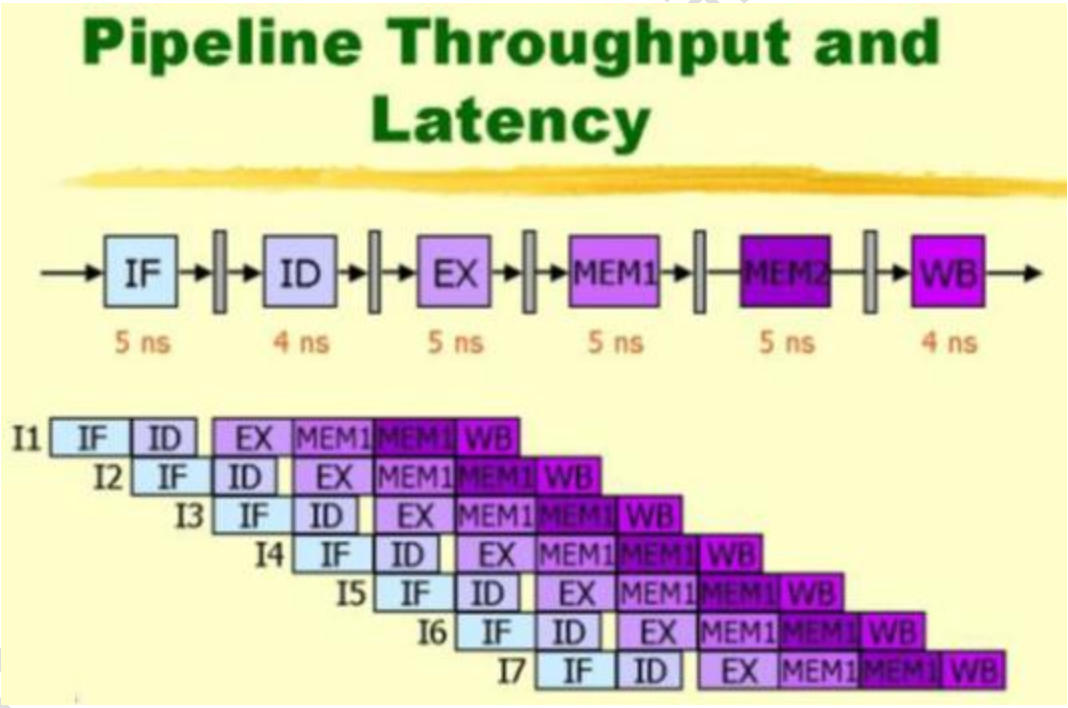
شکل ۲-۷: جریان هوای داخل کانتینر نشان داده شده در شکل ۲-۶. این دیاگرام سطح مقطع دو رک در هر طرف کانتینر را نشان می‌دهد. هوای سرد به داخل راهرو در وسط کانتینر دمیده می‌شود و به سرورها مکیده می‌شود. هوای داغ در لبه‌های کانتینر برمی‌گردد. این طرح جریان هوای گرم و سرد را مجزا می‌کند.

برای سنجش توان پردازشی سیستم ها دو فاکتور کلیدی داریم که عبارتند از:

۱- تاخیر[1]: مدت زمانی که از زمان شروع یک کار تا اولین پاسخ آن طول می کشد.

۲- توان عملیاتی[2]: میزان کار انجام شده در واحد زمان می‌باشد

مثلا در یک پردازنده که برای اجرای هر دستور از k مرحله‌ی با تاخیر t استفاده می‌کند تاخیر اجرای هر دستور kt می‌باشد و زمان اجرای n دستور knt خواهد بود. اگر همین پردازنده بصورت خط لوله[3] سازمان دهی شود زمان اجرای اولین دستور kt خواهد بود ولی زمان اجرای n دستور [k+n-1]t می‌باشد. بنابراین تاخیر اجرای تک دستور در هر دو یکی است ولی توان عملیاتی و میزان کار انجام شده در پردازنده با خط لوله تقریبا k برابر پردازنده بدون خط لوله است. همانطور که در شکل ۲-۸ نشان داده شده است تاخیر اجرای یک دستور ۲۹ پالس است ولی در 64 پالس ۷ دستور اجرا شده است که اگر خط لوله نداشتیم ۷ دستور در 7*28=196 پالس اجرا می شدند بنابراین توان عملیاتی با خط لوله نسبت به بدون خط لوله 196/64=3 می‌باشد. تاخیر مرتبط با تاخیر پردازنده، حافظه نهان، حافظه RAM، و تاخیر دیسک می‌باشد که با افزایش سرعت آنها تاخیر را کاهش می‌دهیم. اما توان عملیاتی فقط با تاخیر مرتبط نیست بلکه به تکنیک‌های مرتبط می‌باشد که برای استفاده از سخت افزارها استفاده می‌کنیم. مواردی که بر توان عملیاتی تاثیر دارند عبارتند از: کارهای محدود به پردازنده، محدود به I/O، محدود به حافظه، محدود به شبکه[4] و ..



شکل ۲-۸: مقایسه تاخیر با توان عملیاتی.

۲-۴ سیستم ذخیره سازی توزیع شده

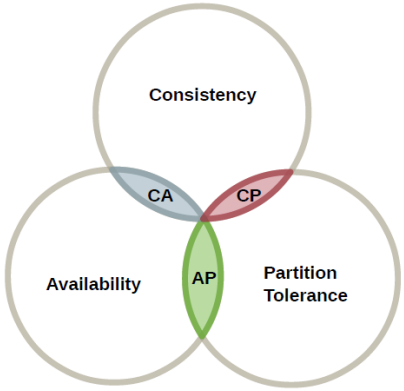
با توجه به مطالب گفته شده برای کلان داده بهتر است از سیستم‌های توزیع شده (مثل ورهاوس‌های گوگل) استفاده کرد. مثلا سوله توضیح داده شده دارای ۴۰۰۰۰ سرور است. سوالاتی که با آنها مواجه می‌شویم این است که فایل‌ها و داده‌های کلان داده را چگونه بنویسیم؟ در کدام

[1] Latancy
[2] Throughput
[3] Pipeline
[4] Compute Bound, Memory Bound, I/O Bound, Netwotk Bound

سرور(ها)؟ چگونه آنها را بخوانیم؟ چگونه از توان پردازشی تعداد زیادی سیستم توزیع شده بطور بهینه استفاده کنیم؟ مدیریت داده‌های کلان در یک سیستم توزیع شده چگونه باید باشد؟ اولین چالش راجع به کلان داده این است که چگونه یک سیستم ذخیره توزیع شده با مقیاس بالا برای نگهداری استراتژیک داده‌ها و پردازش و تجزیه تحلیل موثر داده‌ها توسعه دهیم. برای استفاده از یک سیستم توزیع شده برای ذخیره انبوه داده‌ها، فاکتورهای زیر را باید مد نظر قرار داشت:

- سازگاری[1]: یک سیستم توزیع شده نیازمند این است که چندین سرور بطور همکار داده‌ها را ذخیره کنند، هرچه تعداد سرورها بیشتر باشند، احتمال شکست سرور بیشتر می‌شود. معمولاً داده‌ها به چندین قطعه تقسیم می‌شوند تا در سرورهای مختلف ذخیره شوند تا قابلیت دسترسی را در صورت خطای سرور تضمین کنند. به هر حال شکست‌های سرور و ذخیره موازی ممکن است باعث ناسازگاری در بین کپی‌های مختلف داده‌های مشابه شود. سازگاری یعنی اینکه اطمینان بدهیم کپی‌های چندگانه از داده‌های مشابه، مشابه باشند.
- قابل دسترس بودن[2]: یک سیستم ذخیره توزیع شده در چندین مجموعه سرور عمل می‌کند. هر چه سرورهای بیشتری استفاده شوند، شکست‌های سرور اجتناب ناپذیر خواهد بود. انتظار داریم کل سیستم بطور جدی در مقابل خواندن/نوشتن ترمینال‌های مشتریان متاثر نشود. یا به عبارتی دسترس پذیری به داده‌ها در صورت وجود خرابی امکان پذیر باشد.
- تحمل پذیری پارتیشن[3]: چندین سرور در یک سیستم ذخیره سازی توزیع شده توسط یک شبکه متصل می‌شوند. شبکه می‌تواند شکست‌های لینک/گره یا تراکم‌های موقت داشته باشد. سیستم توزیع شده باید سطح خاصی از تحمل‌پذیری در مقابل شکست‌های شبکه داشته باشد. مطلوب است که ذخیره توزیع شده هنوز کار کند اگر شبکه به چندین پارتیشن شکسته شود.

Eric Brewer در سال ۲۰۰۰ فرضیه[4] CAP را مطرح کرد که مشخص می‌کند یک سیستم توزیع شده بطور همزمان نمی‌تواند نیازهای سازگاری، قابل دسترس بودن، و تحمل پارتیشن را داشته باشد. در سال ۲۰۰۲، اثبات شد که فرضیه CAP درست می‌باشد و تبدیل به نظریه[5] شد. در شکل ۲- ۹ نظریه CAP بصورت گراف نشان داده شده است.



شکل ۲-۹: نظریه CAP

با توجه به شکل ۲-۹ چون سازگاری C، در دسترس بودن A، و تحمل پذیری پارتیشن P بطور همزمان بدست نمی‌آید یا سیستم CA داریم با نادیده گرفتن تحمل پارتیشن، یا یک سیستم CP داریم با نادیده گرفتن قابل دسترس بودن، و یا یک سیستم AP با نادیده گرفتن سازگاری. براساس هدف طراحی یکی از اینها انتخاب می‌شود. این سه سیستم را در ادامه بحث می‌کنیم:

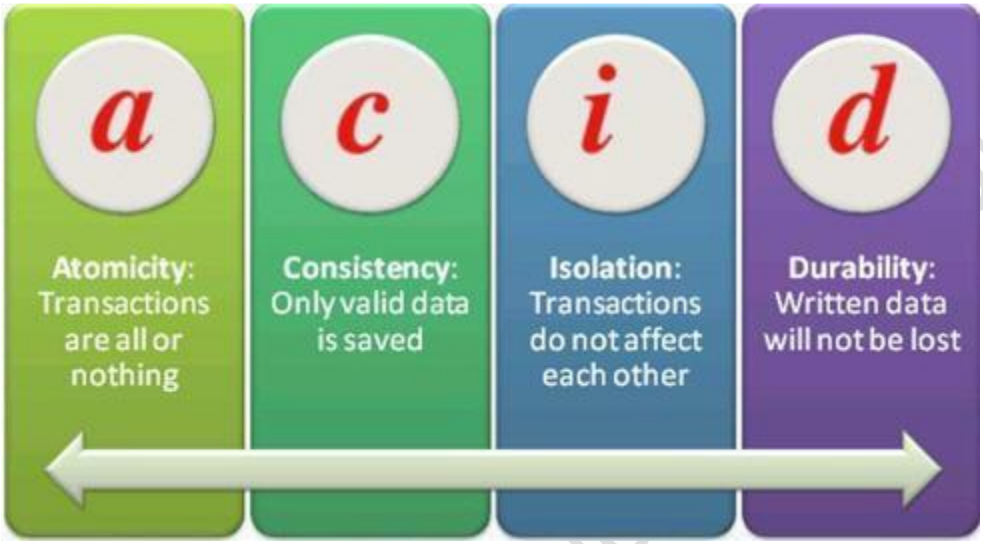
[1] Consistency
[2] Availability
[3] Partition Tolerance
[4] Hypothesis
[5] Theory

- سیستم‌های CA: تحمل‌پذیری پارتیشن را ندارند یعنی نمی‌توانند خطاهای شبکه را مدیریت کنند. بنابراین سیستم‌های CA معمولاً به عنوان سیستم‌های ذخیره سازی با یک سرور به نظر می‌رسند، مثل پایگاه داده‌های رابطه‌ای در مقیاس کوچک سنتی. چنین سیستمی یک کپی از داده‌ها را دارد بنابراین سازگاری به سادگی بدست می‌آید. قابل دسترس بودن با طراحی خوب پایگاه‌های داده رابطه‌ای بدست می‌آید. به هر حال چون سیستم‌های CA نمی‌توانند شکست‌های شبکه را مدیریت کنند، آنها نمی‌توانند برای استفاده چندین سرور توسعه داده شوند. به همین خاطر اکثر سیستم‌های ذخیره مقیاس-بزرگ، سیستم‌های CP یا سیستم‌های AP هستند.
- سیستم‌های CP: در مقایسه با سیستم‌های CA، تحمل‌پذیری پارتیشن را تضمین می‌کنند. بنابراین سیستم CP می‌تواند توسعه داده شود یا یک سیستم توزیع شده شود، سیستم‌های CP معمولاً چندین کپی از داده‌های مشابه را نگه می‌دارند تا سطحی از تحمل‌پذیری خطا را تضمین کنند. سیستم‌های CP همچنین سازگاری را تضمین می‌کنند، مثلاً چندین کپی از داده‌های مشابه تضمین می‌شود که کاملاً مشابه باشند. به هر حال CP نمی‌تواند قابل دسترس بودن را تضمین دهد به خاطر هزینه بالای تضمین سازگاری. بنابراین سیستم‌های CP مناسب برای سناریوهای با بار متوسط هستند اما نیازهای شدید به درستی داده دارند (مثل تجارت داده). BigTable و Hbase دو سیستم CP رایج هستند. BigTable شناخته شده است زیرا برای مدیریت داده‌های زمینه موتور جستجوی گوگل موفق بود. چون اکثر داده‌های گوگل ساختار یافته هستند، BigTable عمدتاً داده‌ها را با جدول ذخیره می‌کند. وقتی مقدار زیادی از اطلاعات در یک جدول قرار داده می‌شوند، اندازه جدول رشد می‌کند. چنین اطلاعاتی باید پارتیشن شوند و بصورت مجزا ذخیره شوند. جدول معمولاً خیلی خلوت می‌باشد بنابراین BigTable ستون‌ها را به چندین خانواده ستون مختلف تقسیم می‌کند که هر خانواده ستون داده‌های از نوع مشابهی را ذخیره می‌کند. بنابراین داده‌ها مشابه با هم ذخیره می‌شوند و اطلاعات با نوع مشابه به روش مشابهی پردازش می‌شوند که آن را برای کاربران سیستم آسان می‌کند. در یک خانواده مشابه، ستون‌های جدید بطور دلخواه و اتفاقی درج می‌شوند بنابراین محدودیت توسعه BigTable را کاهش می‌دهند. BigTable مشابه GFS طراحی شده است که یک رئیس و چند سرورر Tablet بصورت هم‌بندی ستاره در سیستم هستند. ساختار ستاره باعث ایجاد SPOF[1] می‌باشد. بار سرور رئیس به منظور حداقل کردن خطاهای رئیس باید کاهش داده شود. در BigTable، انتقال داده و آدرس‌دهی داده در سیستم رئیس انجام نمی‌شود. بنابراین بار رئیس زیاد نیست. به منظور حل مشکل SPOF، BigTable یک مکانیزمی برای انتخاب رئیس دارد.
- سیستم AP: نیز امکان تحمل پارتیشن را تضمین می‌دهند. اما سیستم‌های AP از سیستم‌های CP متفاوتند زیرا سیستم‌های AP، قابل دسترس بودن را نیز اطمینان می‌دهند. به هر حال سیستم‌های AP فقط سازگاری مشروط[2] را تضمین می‌کنند نه سازگاری قوی دو سیستم قبل را. بنابراین سیستم‌های AP به سناریوهایی که درخواست‌های متناوب وجود دارد اما نه با نیاز به دقت بالا مناسب هستند. به عنوان مثال در سیستم‌های SNS[3] (سرویس‌های شبکه اجتماعی)، ملاقات‌های همروند زیادی برای داده‌ها وجود دارد اما مقدار خاصی از خطاهای داده قابل تحمل است. بنابراین چون سیستم‌های AP سازگاری مشروط را تضمین می‌کنند داده‌های دقیق می‌توانند بعد از مقداری تأخیر بدست آیند. Dynamo, Cassandra دو سیستم AP رایج هستند. Cassandra با قابلیت توسعه صدا، برای ذخیره داده‌های متنی حجیم در شرکت‌های SNS برخط تجاری مثل فیس‌بوک و توییتر استفاده می‌شود.

ویژگی ACID تراکنش‌ها: به تراکنشی ACID[4] می‌گوییم اگر ویژگی‌های اتمیک، سازگاری، ایزوله بودن، بادوام بودن را داشته باشد. ویژگی اتمیک یعنی یا تراکنش‌ها همه انجام می‌شوند یا هیچکدام انجام نمی‌شوند. ویژگی سازگاری یعنی فقط داده‌های معتبر ذخیره می‌شوند.

[1] Single Point of Failure
[2] Eventual Consistence
[3] Social Network Services
[4] Atomicity, Consistency, Isolation, Durbility

ویژگی ایزوله بودن یعنی تراکنش‌ها بر هم تاثیری ندارند. و ویژگی دوام یعنی داده‌های ذخیره شده از بین نخواهند رفت. که این در شکل ۲-۱۰ نشان داده شده است.



شکل ۲-۱۰ تراکنش‌های ACID

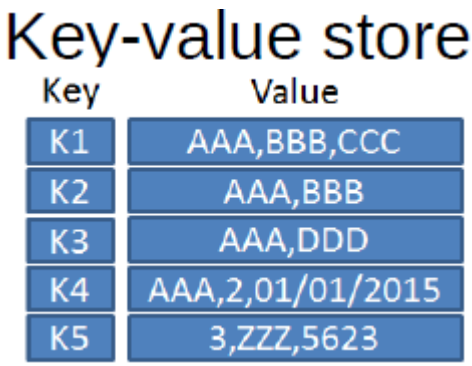
حال که فاکتورهای سنجش روش ذخیره سازی را بیان کردیم در ذخیره سازی روش ذخیره داده‌ها در فایل‌ها یا پایگاه داده‌ها و سیستم‌های فایل توزیع شده و عملیاتی را بررسی خواهیم کرد. سیستم‌های فایل توزیع شده را در فصل سوم بررسی می‌کنیم اما برای ذخیره داده‌ها چند فن‌آوری پایگاه داده داریم که عبارتند از: پایگاه داده‌های کلید-مقدار[۱]، پایگاه داده‌های ستون گرا[۲]، و پایگاه داده‌های سند گرا[۳]، براساس گراف[۴].

فن‌آوری پایگاه داده: فن‌آوری پایگاه داده برای بیشتر از ۳۰ سال تکامل یافته است. سیستم‌های پایگاه داده مختلف برای مدیریت مجموعه داده‌های با مقیاس‌های مختلف و پشتیبانی از کاربردهای مختلف توسعه داده شده‌اند. واضح است که پایگاه داده‌های رابطه‌ای سنتی نمی‌توانند چالش‌های مرتبط با مقیاس کلان داده را برآورده کنند. پایگاه داده‌های NoSQL[۵] (پایگاه داده‌های غیررابطه‌ای) برای ذخیره کلان داده رایج شدند. پایگاه داده‌های NoSQL ویژگی مدهای منعطف، پشتیبانی از کپی ساده، سازگاری مشروط، و پشتیبانی حجم زیاد داده‌ها را دارند. پایگاه داده‌های NoSQL فن‌آوری ساده برای کلان داده می‌باشند. ما چهار پایگاه داده NoSQL عمده را در این بخش دنبال می‌کنیم: پایگاه داده‌های کلید-مقدار پایگاه داده‌های ستون گرا، پایگاه داده‌های سند گرا و براساس گراف که هر کدام براساس مدل داده خاصی هستند.

الف- پایگاه داده‌های کلید-مقدار: پایگاه داده‌های کلید-مقدار از یک مدل ساده تشکیل شده‌اند و داده‌ها براساس کلید-مقدار مرتبط ذخیره می‌شوند. هر کلید منحصر بفرد است و مشتریان ممکن است مقادیر پرس و جو شده را براساس کلیدها وارد کنند. چنین پایگاه داده‌های کلید-مقدار مدرن توسعه پذیری بالا و زمان پاسخ پرس و جوی کمتری نسبت به پایگاه داده‌های رابطه ای

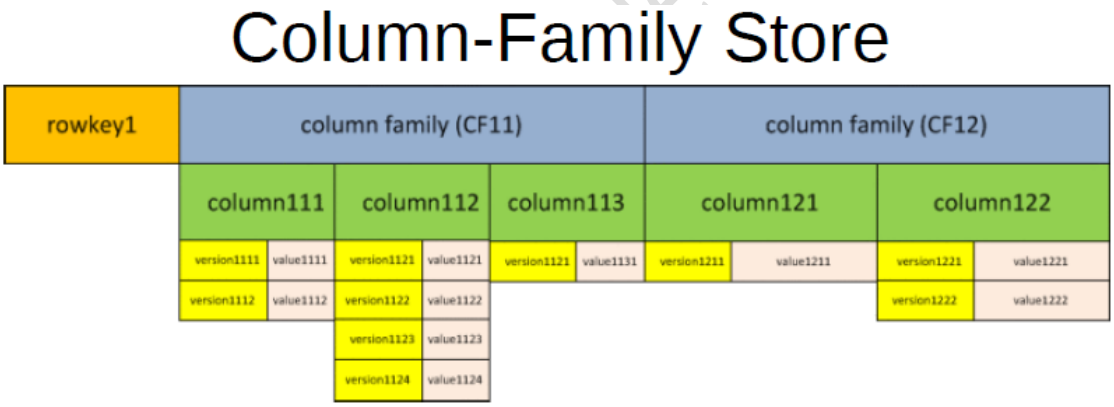
[1] Key-Value
[2] Column-oriented
[3] Document-Oriented
[4] Graph based
[5] Not Only SQL

دارند. در سال‌های اخیر پایگاه داده‌های کلید-مقدار زیادی ظاهر شدند. مثل سیستم Dynamo آمازون. در فصل سوم Dynamo و چند پایگاه داده کلید-مقدار مهم را معرفی می‌کنیم. شکل ۲-۱۱ ساختار پایگاه داده‌های کلید-مقدار را نشان می‌دهد.



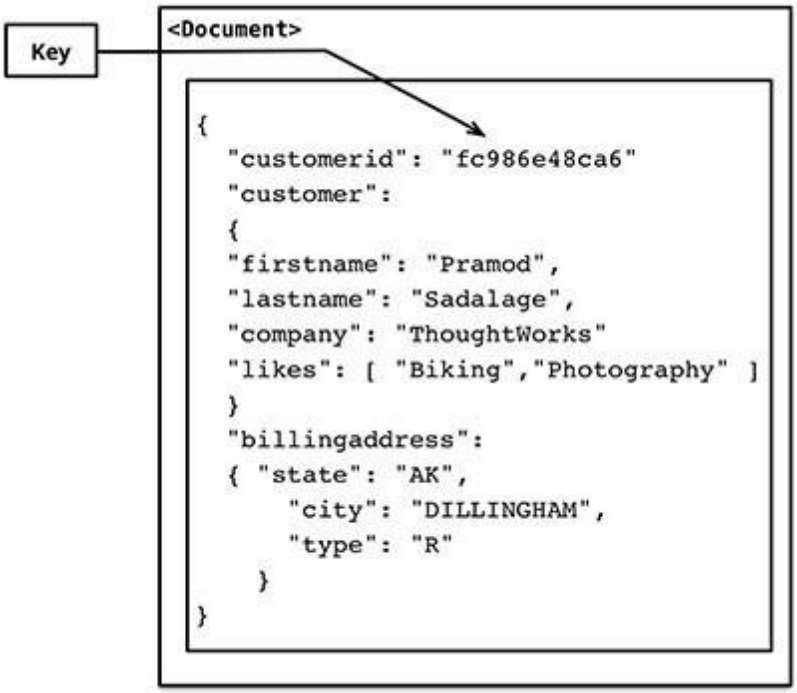
شکل ۲-۱۱ ساختار پایگاه داده‌های براساس کلید-مقدار.

ب-پایگاه داده‌های ستون گرا: در پایگاه داده‌های ستون گرا، داده‌ها بر اساس ستون‌ها به جای سطرها پردازش می‌شوند. ستون‌ها و سطرها به چندین گره تقسیم می‌شوند تا توسعه پذیری برآورده شود. پایگاه داده‌های ستون-گرا عمدتاً توسط BigTable گوگل بررسی می‌شود. که در فصل سوم آن را بررسی می‌کنیم. در شکل ۲-۱۲ ساختار پایگاه داده‌های ستون گرا نشان داده شده است.



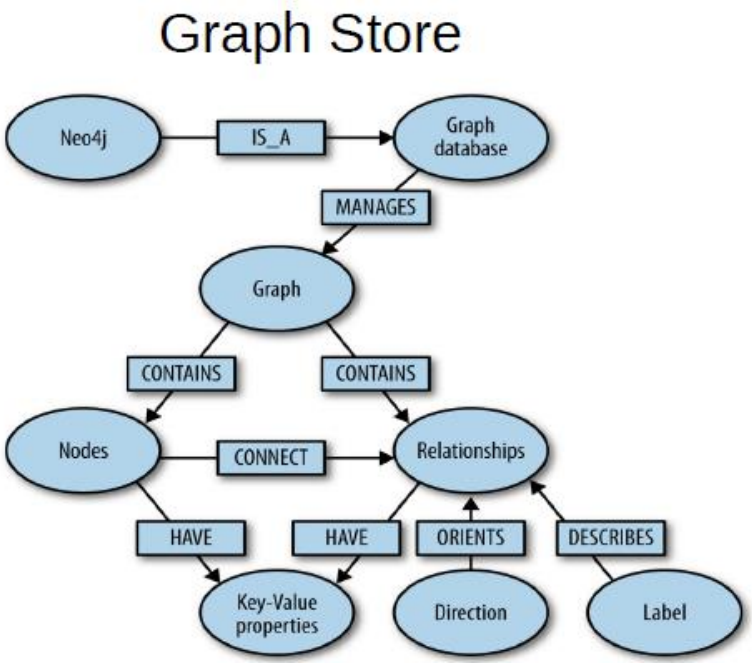
شکل ۲-۱۲ ساختار پایگاه داده‌های ستون‌گرا

ج-پایگاه داده‌های سندی: در مقایسه با ذخیره سازی بر اساس کلید-مقدار، ذخیره‌سازی بر اساس سند می‌تواند شکل‌های پیچیده‌تری از داده را پشتیبانی کند چون اسناد از مدهای strict تبعیت نمی‌کنند، نیازی به مهاجرت مد اتصال نیست. به علاوه زوج‌های کلید-مقدار هنوز می‌توانند ذخیره شوند. مثل MongoDB, SimpleDB, CouchDB. در شکل ۲-۱۳ ساختار پایگاه داده‌های سندی را نشان داده‌ایم. در این نوع داده‌ها می‌توانند در فایل‌های متنی، JSON ، XML و ... ذخیره شوند.



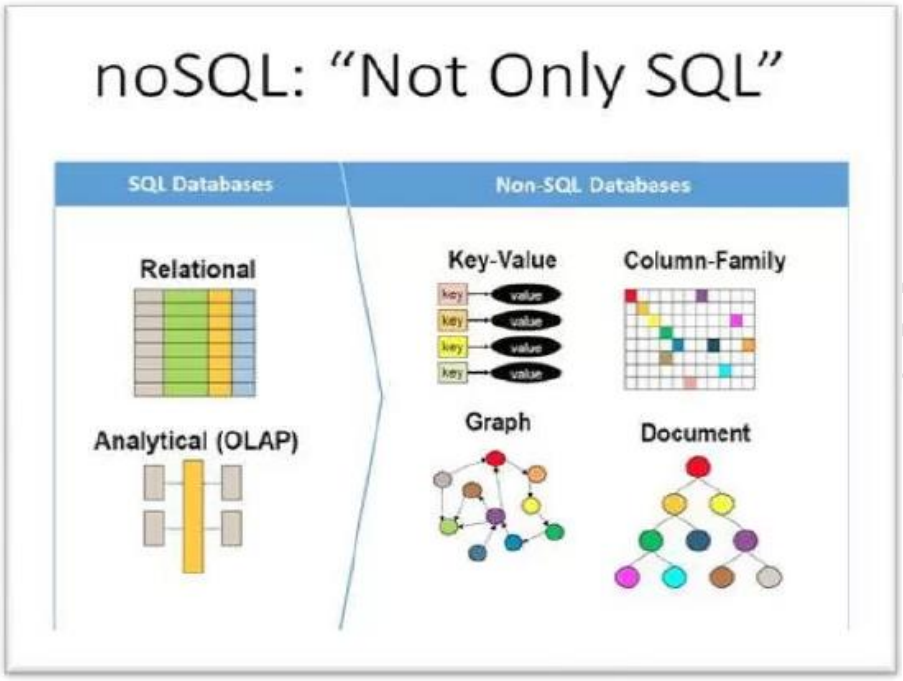
شکل ۲-۱۳ ساختار پایگاه داده‌های سندگرا

د-ذخیره سازی براساس گراف: در شبکه های اجتماعی و شبکه‌های پیچیده برای تجزیه تحلیل ارتباطات بین افراد، اشیاء، اجتماعات، و ... از روش برپایه گراف استفاده می‌شود و در آنها هدف اصلی استخراج اجتماعات از داده‌ها، بررسی استحکام اجتماعات، انتشار ویروس‌ها در شبکه و جلوگیری از آنها و جلوگیری از شکست‌های آبشاری آنها می‌باشد. در شکل ۲-۱۴ ساختار این روش نشان داده شده است. در این گراف ها هر شیء یک گره و ارتباط بین آنها یک یال می‌باشد.



شکل ۲-۱۴: ساختار ذخیره سازی داده‌ها به روش گراف.

به طور خلاصه روش‌های مختلف ذخیره سازی سنتی (پایگاه‌های رابطه‌ای) و NOSQL در شکل ۲-۱۵ نشان داده شده است.



شکل ۲-۱۵: خلاصه روش‌های ذخیره سازی داده‌ها

فصل سوم سیستم‌های فایل توزیع شده

در این فصل ساختار سیستم‌های توزیع شده شامل HDFS,BigTable, Dynamo, Cassandra را توضیح می‌دهیم.

۱-سیستم‌های فایل توزیع شده Hadoop: هدوپ از دو بخش کلیدی تشکیل شده است: **HDFS, MapReduce**

برای اینکه بفهمیم چگونه می توانیم یک خوشه هدوپ را به صدها حتی هزاران گره مقیاس پذیر تبدیل کنیم اول باید سیستم فایل توزیع شده آن یا HDFS[۱] را بفهمیم. داده ها در یک خوشه[۲] هدوپ به قطعات کوچکتر شکسته می شوند (که به آنها بلوک می‌گویند) و در طول خوشه توزیع می‌شوند. در این روش توابع MapReduce می توانند به مجموعه کوچکتری از داده‌های کلان شما اعمال شوند و این مقیاس پذیری[۳] لازم را برای پردازش کلان داده فراهم می‌کند.

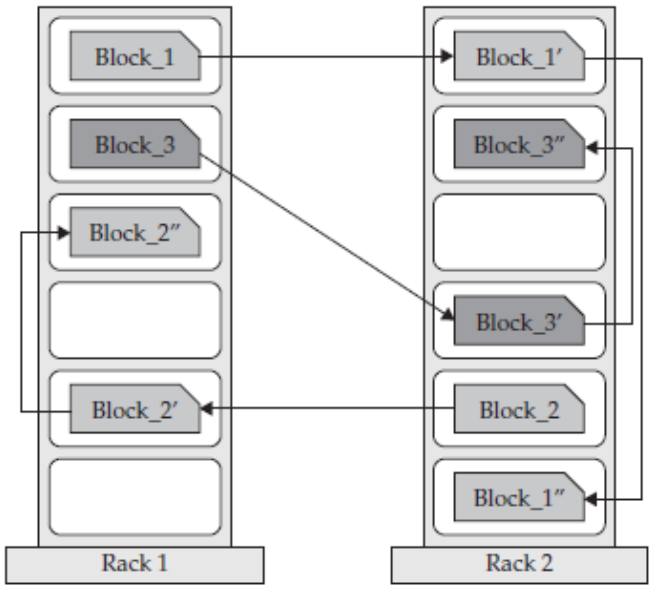
هدف هدوپ استفاده از سرورهای قابل دسترس[۴] مشترک در یک خوشه خیلی بزرگ می باشد، که هر سرور مجموعه ای از دیسک درایوهای داخلی ارزان دارد. برای کارایی بیشتر MapReduce سعی می‌کند تا بارهای کاری[۵] را به این سرورها که داده‌هایی که باید پردازش شوند در آنها ذخیره می‌شوند منتسب کند. به این محلیت داده[۶] می‌گویند.

در هدوپ SAN[۷] (دستگاه‌های ذخیره شبکه) یا دستگاه‌های ذخیره سازی الحاقی شبکه(NAS[۸]) توصیه نمی‌شود. زیرا در SAN یا NAS سربارهای ارتباطی شبکه می‌تواند باعث گلوگاه کارایی شود بالاخص برای خوشه‌های بزرگتر. یک لحظه خوشه‌ای از ۱۰۰۰ ماشین را در نظر بگیرید که هر ماشین سه عدد دیسک داخلی دارد. بنابراین نرخ خطای خوشه‌ای با ۳۰۰۰ دیسک درایو ارزان + ۱۰۰۰ سرور را محاسبه کنید. هر چه تعداد سیستم‌ها بیشتر شود MTTF[۹] بیشتر می‌شود. بنابراین باید مکانیزمی را ایجاد کرد که کل سیستم بدون توجه به عملکرد تک تک اجزا بطور مطمئن کار کند. MTTF در یک خوشه هدوپ شبیه زیپ جیب کاپشن شماست: ممکن است دچار خطا شود (وقتی خطا می‌دهد که شما به آن نیاز دارید). نقطه قوت هدوپ این است که تحمل پذیری خطای داخلی دارد و قابلیت جبران مافات دارد. این در HDFS نیز وجود دارد. زیرا داده‌ها به بلوک‌هایی شکسته می‌شوند و کپی‌های این بلوک‌ها در سرورهای دیگر که در خوشه هدوپ هستند ذخیره می‌شوند. یعنی یک فایل منفرد دقیقاً به بلوک‌های کوچکتر تقسیم می‌شود که در چندین سرور در کل خوشه تکرار می‌شوند.

فایلی را در نظر بگیرید که شماره تلفن‌های هر فردی در ایران را دارد. افراد با حرف اول فامیل الف ممکن است در سرور ۱ ذخیره شود، ب در سرور ۲ و غیره. در دنیای هدوپ قطعات این شماره تلفن‌ها در یک خوشه ذخیره می‌شوند و برای بازسازی کل شماره تلفن‌ها، برنامه شما نیازمند این است که بلوک‌ها را از همه سرورها در خوشه جمع‌آوری کند. برای رسیدن به قابل دسترس بودن در صورت خرابی اجزاء، HDFS قطعات کوچکتر را در دو سرور اضافی بصورت پیش‌فرض ذخیره می‌کند (کپی می‌کند[۱۰]). که در شکل ۳-۱ نشان داده شده است:

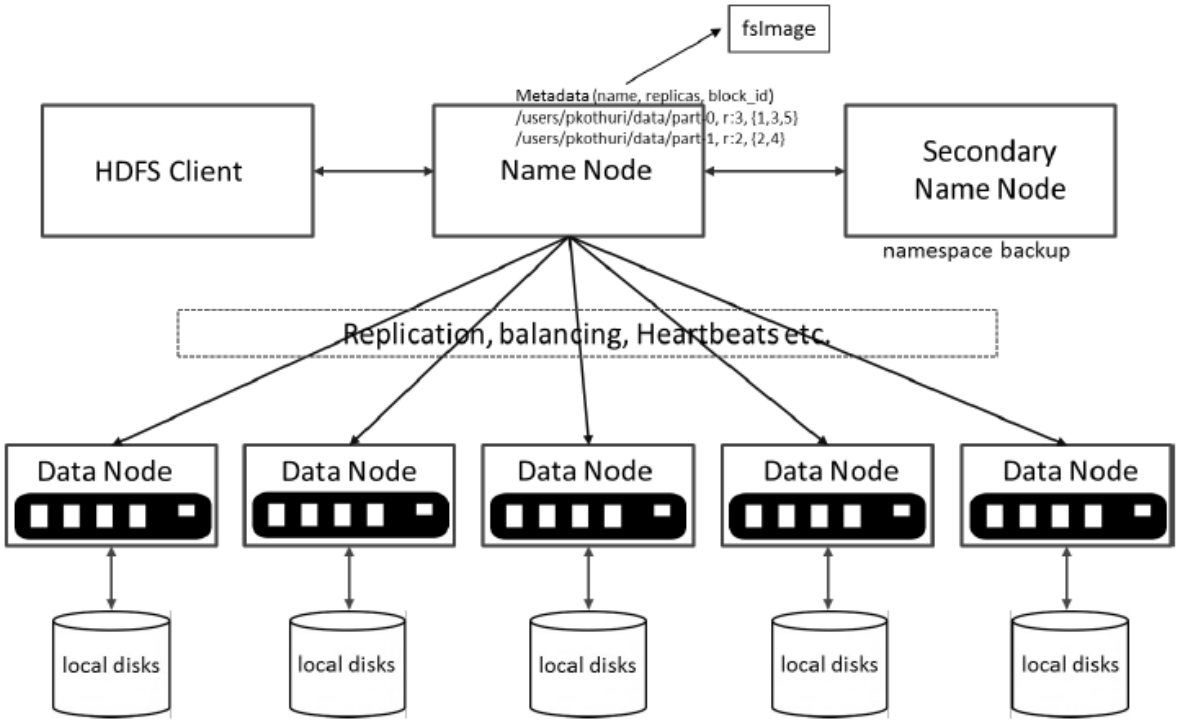
این افزونگی چندین مزیت دارد: مهمترین آن قابلیت دسترس بالاتر است. مزیت دیگر آن مقیاس‌پذیری و محلیت داده می‌باشد که وقتی با مجموعه داده بزرگ کار می‌کنید، بحرانی می‌باشد.

[1] Hadoop Distributed File System
[2] Cluster
[3] Scalability
[4] Available
[5] Workloads
[6] Data Locality
[7] Storage Area Network
[8] Network attached Storage
[9] Mean time to failure
[10] Replica



شکل ۳-۱: روش ذخیره سازی فایل‌ها در HDFS. هر بلوک در سه سرور نوشته می‌شوند و برای افزونگی حداقل یک بلوک در سروری در رک دیگری نوشته می‌شود.

معماری HDFS در شکل ۳ نشان داده شده است:



شکل۳-۲: معماری HDFS: در HDFS یک NameNode و تعدادی DataNode داریم.

یک فایل داده در HDFS به بلوک‌هایی تقسیم می‌شود، و اندازه پیش‌فرض این بلوک‌ها برای Apache Hadoop، 64MB می‌باشد. برای فایل‌های بزرگتر ، اندازه بلوک بزرگتر ایده‌ی خوبی می‌باشد. زیرا این مقدار متادیتای مورد نیاز برای NameNode را خیلی کاهش می‌دهد. در BigInsights، اندازه پیش‌فرض برای بلوک 128MB می‌باشد زیرا در تجارب کاری IBM Hadoop، رایجترین استفاده‌ها، فایل‌ها و بارهای کاری بزرگتر با خواندن‌های ترتیبی می‌باشد. یادآوری می‌کنیم که Hadoop برای پیمایش مجموعه داده‌های بزرگتر طراحی شده است، بطوریکه هر سرور با قطعات بزرگتری از داده‌ها بطور همزمان کار کند.

هماهنگی در یک خوشه سربار قابل توجهی دارد، بنابراین قابلیت پردازش قطعات بزرگتر داده بطور محلی بدون ارسال به سایر گره‌ها به بهبود کارایی و سربار مربوط به نرخ کار واقعی کمک می‌کند. یادآوری می‌کنیم که هر بلوک داده بطور پیش‌فرض در سه سرور مختلف ذخیره می‌شود. در Hadoop این عمل توسط HDFS انجام می‌شود تا اطمینان بدهد که حداقل دو بلوک در سرورهای رک‌های مختلف ذخیره می‌شوند تا در صورت خرابی کامل یک رک قابلیت اطمینان را گارانتی کند.

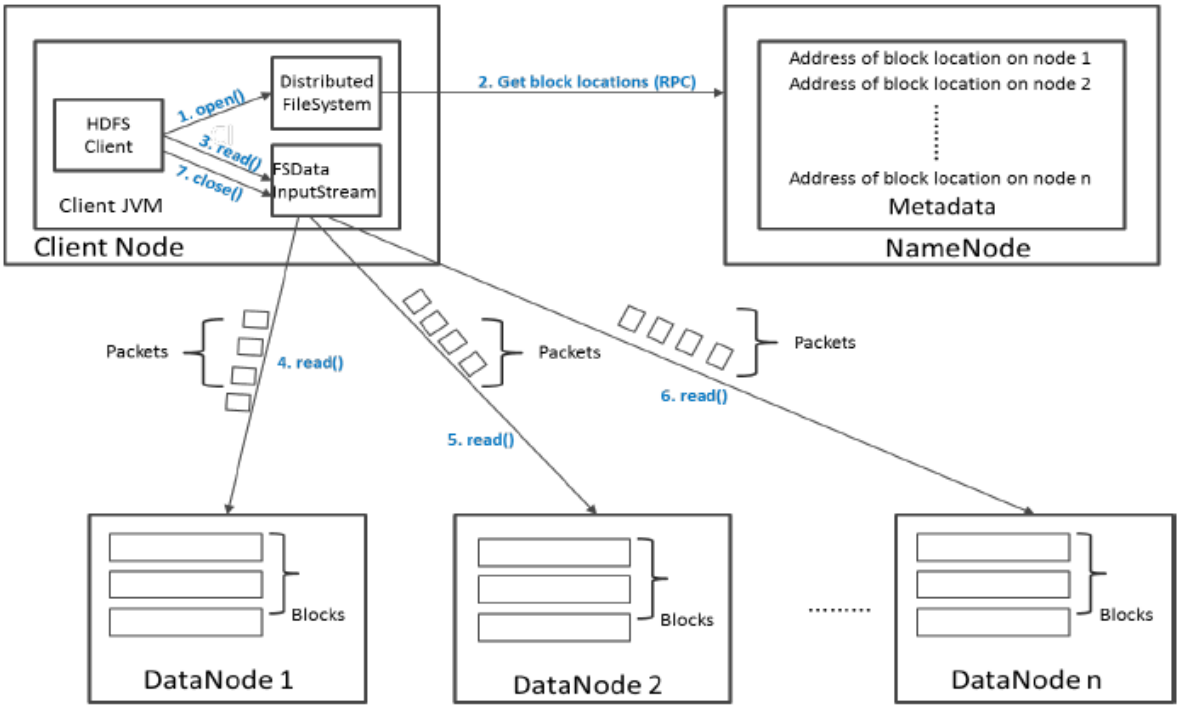
کل منطق جایگزینی داده هدوپ با یک سرور خاص مدیریت می‌شود که NameNode نامیده می‌شود. این سرور NameNode همه داده‌های HDFS را ردیابی می‌کند، مثل اینکه داده‌ها کجا ذخیره می‌شوند و غیره. همه اطلاعات NameNode در حافظه (memory) ذخیره می‌شوند، که باعث می‌شود زمان‌های پاسخ سریع به دستکاری اطلاعات یا درخواست‌های خواندن داشته باشیم. حال می‌دانیم شما به چه فکر می‌کنید: اگر فقط یک NameNode برای کل خوشه هدوپ وجود داشته باشد، ذخیره این اطلاعات در حافظه یک نقطه شکست[1] ایجاد می‌کند. به همین دلیل قویاً توصیه می‌کنیم که اجزاء سروری که شما برای NameNode انتخاب می‌کنید از بقیه سرورهای خوشه‌های هدوپ قویتر باشد تا احتمال شکست‌ها را حداقل کند. به علاوه قویاً توصیه می‌کنیم از متادیتاهای ذخیره شده در NameNode پشتیبان‌گیری کنید. ناپدید شدن داده در این متا دیتا باعث گم شدن دائمی داده‌های مربوط در خوشه می‌شود. البته راه دیگری که هدوپ نگارش 0.21 ایجاد کرد، ایجاد یک سرور پشتیبان برای این NameNode به نام Secondary NameNode می‌باشد.

همانطور که در شکل ۳-۱ مشاهده می‌کنید یک فایل داریم که از سه بلوک داده ساخته شده است که بلوک داده (نشان داده شده با Block-n) در دو سرور دیگر تکرار می‌شود. (Block-n', Block-n'')، تکرار دوم و سوم در یک رک فیزیکی مجزا در گره‌های مجزا برای حفاظت بیشتر ذخیره می‌شوند.

نکته مهم راجع به اپلیکیشن Hadoop MapReduce این است که برخلاف تکنولوژی‌های گرید قدیمی، توسعه دهندگان نباید با مفهوم NameNode و جاییکه داده‌ها ذخیره می‌شوند آشنا باشند. اینکار را هدوپ برای شما انجام می‌دهد. هنگامی که از MapReduce استفاده می‌کنید هدوپ به NameNode متصل می‌شود و سرورهایی که بخش‌های داده را نگه می‌دارند را یافته و اپلیکیشن شما را به آن گره‌ها می‌فرستد تا بطور محلی در آن اجرا شوند. بطور مشابه وقتیکه شما یک فایل را ایجاد می‌کنید، HDFS بطور خودکار با NameNode ارتباط برقرا می‌کند تا فضای ذخیره سازی را در سرورهای خاص تخصیص دهد و تکرار داده‌ها را انجام دهد.

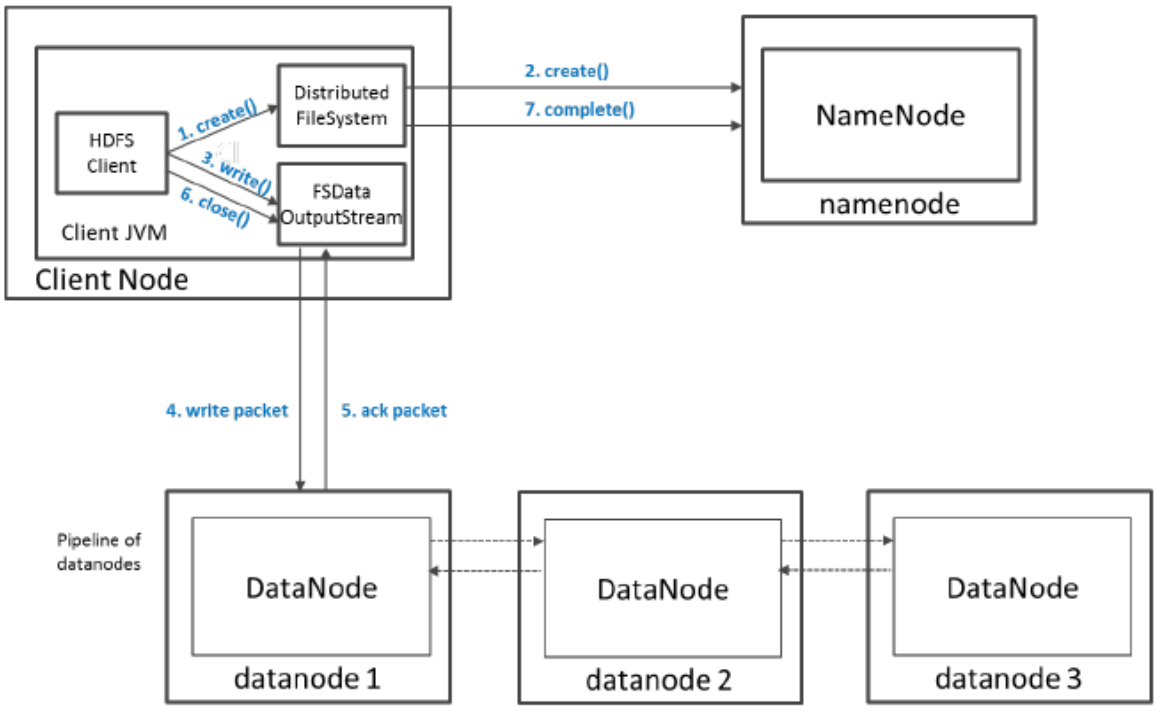
عمل خواندن در HDFS: در شکل۳-۳ بلوک دیاگرام عمل خواندن در HDFS نشان داده شده است.کلاینت برای خواندن یک فایل دستور open را اجرا می‌کند و NameNode مکان ذخیره سازی بلوک‌های فایل را برمی‌گرداند و کلاینت دستور read را اجرا می‌کند و داده‌ها از سرورهایی که NameNode مشخص کرده است بصورت بلوکی خوانده می‌شوند و در انتها کلاینت دستور close را اجرا می کند. در حقیقت کلاینت فقط دستورهای open-read-close را اجرا می‌کند و این HDFS است که بقیه کارها را در اختیار او قرار می‌دهد.

[1] Single Point of Failure (SPOF)



شکل ۳- ۳: خواندن در HDFS

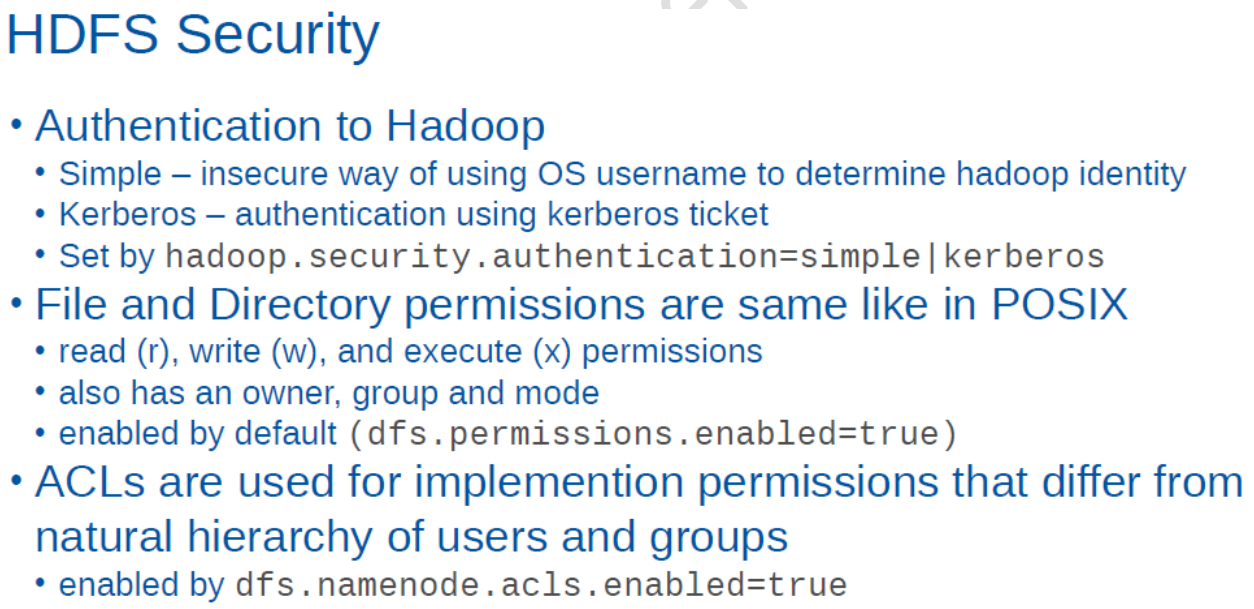
عمل نوشتن در HDFS: شکل ۳-۴ روش نوشتن در HDFS را نشان می‌دهد:



شکل ۳- ۴: عمل نوشتن در HDFS

فرض اولیه HDFS این هست که نوشتن یکبار انجام می شود و خواندن چندین بار انجام می‌شود (write-once-read-many). معمولاً وقتی می‌خواهیم فایل را تغییر دهیم فایل عوض نمی‌شود بلکه فایل جدید نوشته می‌شود و به عبارتی فایل read-only است. بنابراین برای نوشتن کلاینت اول دستور create را اجرا می کند و یک فایل را ایجاد می‌کند و با NameNode ارتباط برقرار می‌کند. سپس دستورات write را صدا می‌زند و بلوک در Datanode ذخیره می شود و خود DataNode تکرار (Replica) ها را ایجاد می‌کند و به کلاینت acknowledge می‌دهد. در انتها دستور complete به NameNode داده شده و مکان Replica ها ذخیره می‌شود. خود سیاست write-once-read many در عمل خواندن تاثیر گذار هست و هنگام خواندن نیاز نیست هر سه کپی داده را بخوانیم اولی را که خواندیم در صورت درست بودن کار خاتمه می‌یابد در صورت غلط بودن به کپی‌های دوم و بعد سوم مراجعه می‌شود. اما در هنگام نوشتن Datanode1 مسئول نوشتن کپی ها در سرورهای افزونه است. نکته دیگر این هست که اگر فایل بزرگ باشد و به چندین shard تقسیم شده باشد لزوماً همه بخش‌ها در یک DataNode نوشته نمی‌شوند. هر DataNode مسئول Replicaهای خودش است و ممکن است هر Shard در یک DataNode ذخیره شود. مثلا اگر فایل ۱۰۰ شارد داشته باشد ممکن است ۱۰۰ عدد datanode1 داشته باشیم که هر کدام یک شارد را ذخیره و کپی آن را در دو سرور دیگر ذخیره خواهند کرد. NameNode مسئول ذخیره سازی فهرست ذخیره هست که فایل‌ها کجا ذخیره شده‌اند.

امنیت HDFS: شکل ۳- ۵ امنیت در HDFS را نشان می‌دهد. برای امنیت نیز package هایی وجود دارد که امنیت آن را تامین می‌کند که یکی از آنها Kerberos می‌باشد. هر فایلی را که می‌خواهیم از HDFS بخوانیم قبلاً اعتبار سنجی شده است (Authenticate). مجوزهای دسترسی نیز مثل لینوکس است که مجوزهای خواندن، نوشتن، اجرا و نویسنده، گروه و mode را شامل می‌شود.



شکل ۳-۵: امنیت در HDFS

بطور خلاصه HDFS یک سیستم فایل توزیع شده برای ذخیره سازی داده‌ها می‌باشد که چون فایل‌ها بصورت بلوک به بلوک ذخیره می‌شوند می‌توان با استفاده از MapReduce این بلوک‌ها را به کارهای map مجزا داد و علاوه بر اینکه حجم حافظه ذخیره سازی را بالا برد سرعت پردازش بطور موازی را نیز به تناسب افزایش داد. البته افزایش سرعت به اندازه افزایش حجم نیست رابطه بین آنها زیرخطی(sublinear) است. یعنی اگر فایل در ۱۰۰ گره داده متفاوت ذخیره شود اجرای موازی MapReduce روی آنها باعث افزایش سرعت ۱۰۰ برابری نیست. مقداری از آن کمتر می‌باشد. همچنین HDFS مقیاس‌پذیری Horizontal دارد یعنی امکان اضافه شدن DataNode و بالانس مجدد آنها را دارد.

اساس MapReduce: MapReduce (MR) قلب هدوپ است. MR یک نمونه برنامه‌نویسی است که مقیاس‌پذیری انبوهی را در صدها یا هزاران سرور در یک خوشه هدوپ می‌دهد. فهم MR برای افرادی که با راه‌حل‌های پردازش داده خوشه‌ای آشنا هستند ساده می‌باشد. عبارت MapReduce به دو کار مجزا و متفاوت اشاره دارد که برنامه‌های هدوپ انجام می‌دهند. کار اول Map می‌باشد که یک مجموعه داده را دریافت می‌کند و به مجموعه داده دیگری نگاشت می‌دهد که عناصر منفرد به دو زوج (key-value) شکسته می‌شوند. کار Reduce گرفتن خروجی Map به عنوان ورودی است و ترکیب آن زوج داده‌ها به مجموعه زوج‌های کوچکتر است. یعنی کار Reduce بعد از کار Map می‌باشد.

یک مثال ساده را در نظر بگیرید. فرض کنید ۵ فایل داریم که هر کدام شامل دو ستون (key-value) می‌باشد که یک شهر و درجه حرارت مرتبط با آن در روزهای متفاوت ضبط شده است. علت بیان این مثال ساده راحتی ردیابی آن می‌باشد. حتماً می‌دانید که کاربردهای واقعی اینقدر ساده نیستند زیرا ممکن است میلیون‌ها یا حتی میلیاردها سطر داشته باشند و ممکن است سطرهای مرتب نداشته باشند. مهم نیست مقدار داده‌هایی که شما نیاز دارید آنالیز کنید چقدر باشد، مفهوم کلیدی که ما در اینجا بیان می‌کنیم یکسان است. در این مثال city کلید و temperature مقدار است. نمونه داده خروجی زیر را در نظر بگیرید:

Toronto, 20
Whitby, 25
NewYork, 22
Rome,32
Toronto, 4
Rome, 33
NewYork,18

جدای از همه داده‌هایی که ما جمع‌آوری کرده‌ایم، می‌خواهیم حداکثر دمای هر شهر را در فایل‌های داده بیابیم (نکته اینکه در هر فایل ممکن است یک شهر چندین بار نشان داده شده باشد). با استفاده از MapReduce، ما می‌توانیم این کار را به ۵ کار map بشکنیم، که هر map رو یکی از ۵ فایل کار کند و کار mapper حرکت در داده‌ها و برگرداندن حداکثر درجه حرارت هر شهر است. به عنوان مثال، نتایج تولید شده از کار mapper داده‌های فوق شبیه این است:

(Toronto,20) , (Whitby, 25), (NewYork, 22), (Rome,33)

فرض کنید خروجی چهار کار mapper دیگر روی ۴ فایل دیگر بصورت زیر باشد:

(Toronto,18) , (Whitby, 27), (NewYork, 32), (Rome,37)
(Toronto,32) , (Whitby, 20), (NewYork, 33), (Rome,38)
(Toronto,22) , (Whitby, 19), (NewYork, 20), (Rome,31)
(Toronto,31) , (Whitby, 22), (NewYork, 19), (Rome,33)

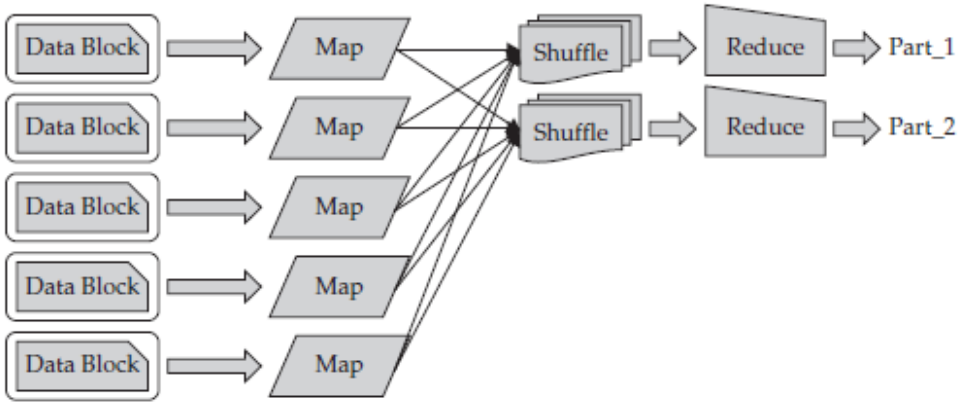
همه این ۵ رشته خروجی به کار reduce خورانده می‌شوند که نتایج ورودی را ترکیب می‌کند و یک زوج شهر–دمای تک برای هر شهر ایجاد می‌کند و نتیجه خروجی زیر را تولید می‌کند.

(Toronto,32) , (Whitby, 27), (NewYork, 33), (Rome,38)

مثال سرشماری را در نظر بگیرید: کار map انجام سرشماری‌های مجزا در هر شهر است و خروجی آن تعداد افراد شهرها می‌باشد و سپس خروجی تمامی شهرها به reduce داده می‌شود و جمع تعداد افراد را به عنوان جمعیت کشور می‌دهد. آیا این بهتر است که شهرهای مختلف به صورت موازی سرشماری شوند یا بصورت سری؟ برنامه Mapreduce به یک job اشاره می‌کند، یک job بطور متوالی به taskهای کوچکتر شکسته می‌شود.

یک اپلیکیشن یک job را به یک گره خاص در خوشه هدوپ تقدیم می‌کند، که یک jobtracker را اجرا می‌کند. Jobtracker با namenode ارتباط برقرار می‌کند تا همه داده‌های مورد نیاز برای این job که موجود در خوشه است را بیابد و job را به کارهای MapReduce برای هر گره می‌شکند تا در خوشه کار کند. این task ها روی گره‌هایی از خوشه که داده‌ها موجودند زمانبندی می‌شوند. نکته اینکه یک task ممکن است به گرهی داده شود که داده‌های مورد نیاز در آن موجود نباشند. در این حالت، گره باید برای داده‌های مورد نیاز از طریق شبکه اتصالی جستجو کند تا task را انجام دهد. قطعاً این مفید نیست بنابراین jobtracker سعی می‌کند از این اجتناب کند و تلاش کند taskها را طوری زمانبندی کند که داده‌ها ذخیره شده‌اند. این مفهوم محلیت داده است که راجع به آن صحبت کردیم و هنگامیکه با حجم زیادی از داده‌ها کار می‌کنیم بحرانی است. در یک خوشه هدوپ، مجموعه ای از daemon هایی که متوالیا در حال اجرا هستند و به آنها عامل jobtracker می‌گویند وضعیت هر task را مانیتور می‌کنند. اگر یک task خطا بدهد وضعیت آن را به jobtracker گزارش می‌دهند و jobtracker مجدداً آن کار را در سایر گره‌های خوشه زمانبندی می‌کند. (شما می‌توانید تعیین کنید که چقدر یک کار می‌تواند تلاش کند قبل از اینکه کل job لغو شود).

شکل ۳-۶ نمونه‌ای از جریان MapReduce را نشان می‌دهد. شما می‌توانید ببینید که چندین کار Reduce می‌توانند اجرا شوند تا موازات را افزایش داده و کارایی را بهبود دهند. در شکل ۶ خروجی map، مستقیماً (با کلید-مقدار)، به reduce task مناسب هدایت می‌شوند. اگر ما حداکثر دما را به عنوان مثال این شکل در نظر بگیریم همه رکوردهایی که مقدار-کلید Toronto را دارند باید به یک کار reduce ارسال شوند تا نتیجه درست بدست آید. (یک reducer باید قادر به دیدن همه دماهای Toronto باشد تا حداکثر را برای شهر مشخص کند). این هدایت رکوردها به کارهای Reduce را بُرزدن (Shuffle) می‌گویند، که ورودی‌ها را از کارهای map می‌گیرد و به کار reduce مناسب می‌فرستد. هدوپ به شما این انتخاب را می‌دهد تجمع محلی روی خروجی هر کار map را انجام دهید قبل از ارسال نتایج به reduce task از طریق تجمع محلی به نام combiner. واضح است اگر چندین کار reduce انجام شوند سربار زیاد خواهد شد اما برای مجموعه داده‌های بزرگ این عمل در کل ،کارایی را افزایش می‌دهد.



شکل ۳- ۶: جریان داده در MapReduce Job

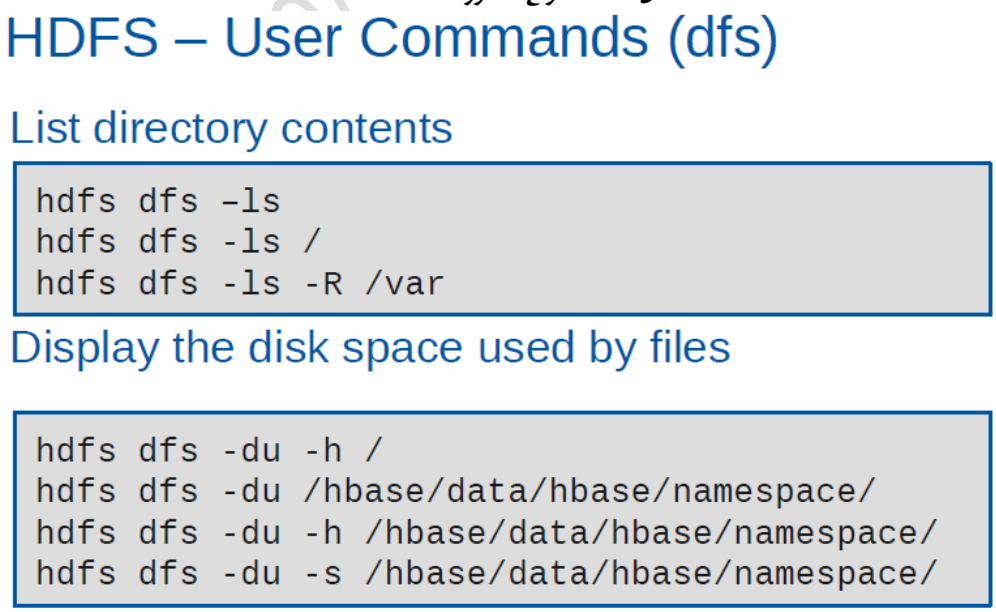
همه برنامه‌های MapReduce که تحت هدوپ اجرا می‌شوند با زبان جاوا نوشته می‌شوند، و فایل جاوا (jar) در jobtracker در گره‌های خوشه هدوپ مختلف توزیع می‌شود تا کارهای map , Reduce را اجرا کند. برای خواندن مطالب بیشتر روی MapReduce می‌توانید اسناد Apache Hadoop را ببینید تا نمونه‌های کدهای جاوای اجرایی را مشاهده کنید.

اگر می‌خواهید سریعترین و ساده‌ترین راه را برای اجرای هدوپ پیدا کنید BigDataUniversity.com را چک کنید و Info BigInsights را دانلود کنید. www.ibm.com/software/dta/infosphere/biginsights.basic.html که به شما قابلیت‌های افزونه آی بی ام را می‌دهد و ۱۰۰ درصد رایگان است.

اجزاء رایج هدوپ

اجزاء رایج هدوپ مجموعه ای از کتابخانه‌ها هستند که زیر پروژه‌های مختلف هدوپ را پشتیبانی می‌کنند. در این بخش شل سیستم فایل را بحث می‌کنیم. HDFS سازگار با سیستم فایل POSIX نمی‌باشد (یونیکس یا لینوکس). برای محاوره با فایل‌های HDFS شما نیاز دارید با رابط فرمان دستورات /bin/hdfs dfs <args> کار کنید که <args> آرگومان‌های فرمانی را نشان می‌دهد که شما می‌خواهید روی فایل‌های سیستم انجام دهید. در شکل ۳–۷ انواع دستوارت HDFS و در شکل‌های ۳–۸–۹ و ۱۰ چند مثال از فرمان‌های شل HDFS را نشان می‌دهیم.

شکل ۳–۷– انواع دستورات HDFS shell



شکل ۳–۸– چند نمونه دستور HDFS

Here are some examples of HDFS shell commands:

cat	Copies the file to standard output (`stdout`).
chmod	Changes the permissions for reading and writing to a given file or set of files.
chown	Changes the owner of a given file or set of files.
copyFromLocal	Copies a file from the local file system into HDFS.
copyToLocal	Copies a file from HDFS to the local file system.
cp	Copies HDFS files from one directory to another.
expunge	Empties all of the files that are in the trash. When you delete an HDFS file, the data is not actually gone (think of your MAC or Windows-based home computers, and you'll get the point). Deleted HDFS files can be found in the trash, which is automatically cleaned at some later point in time. If you want to empty the trash immediately, you can use the `expunge` argument.
ls	Displays a listing of files in a given directory.
mkdir	Creates a directory in HDFS.
mv	Moves files from one directory to another.
rm	Deletes a file and sends it to the trash. If you want to skip the trash process and delete the file from HDFS on the spot, you can use the `-skiptrash` option of the `rm` command.

شکل ۳- ۹- چند نمونه فرمان HDFS

HDFS – User Commands (dfs)

Copy data to HDFS

```
hdfs dfs -mkdir tdata
hdfs dfs -ls
hdfs dfs -copyFromLocal tutorials/data/geneva.csv tdata
hdfs dfs -ls -R
```

```
cd tutorials/data/
hdfs dfs -copyToLocal tdata/geneva.csv geneva.csv.hdfs
md5sum geneva.csv geneva.csv.hdfs
```

شکل ۳- ۱۰- چند نمونه دستور HDFS

توسعه اپلیکیشن در هدوپ

پلت فرم هدوپ می‌تواند ابزاری قدرتمند برای دستکاری مجموعه داده‌های فوق‌العاده بزرگ ارائه دهد. اما MapReduce هدوپ براساس جاوا است و نیازمند برنامه نویسی ماهرانه است. به علاوه برای برنامه نویسان مشکلتر است تا اپلیکیشن‌های MapReduce را برای مشاغلی که پردازش خطالوله‌ای و طولانی می‌خواهند توسعه دهند. در این بخش دو گزینه Pig, Hive را توضیح می‌دهیم.

Pig and Pig Latin: Pig ابتدا در یاهو توسعه داده شد. تا به افراد اجازه دهد از هدوپ استفاده کنند و روی آنالیز مجموعه داده بزرگ تمرکز کنند و زمان کمتری برای نوشتن برنامه‌های mapper و reducer صرف کنند. شبیه خوک‌های واقعی که تقریباً هر چیزی را می‌خورند، زبان برنامه‌نویسی Pig برای این توسعه داده شد تا هر نوع داده‌ای را مدیریت کند. بنابراین Pig از دو بخش ساخته شده است: اولی خود زبان هست که به آن Pig Latin می‌گویند و دوم محیط اجرایی است که برنامه‌های PigLatin در آن اجرا می‌شوند. (مثل رابطه بین JVM و اپلیکیشن جاوا). در این بخش به کل آن Pig می‌گوییم.

اجازه دهید اول به خود برنامه نویسی تمرکز کنیم تا بفهمید از نوشتن mapper و reducer ساده‌تر است. اولین مرحله در برنامه Pig، Load کردن داده‌هایی است که می‌خواهید آنها را از HDFS دستکاری کنید. سپس شما داده‌ها را از طریق تبدیلاتی اجرا می‌کنید (که به مجموعه‌ای از mapper و reducer نگاشت می‌شوند). در انتها شما داده‌ها را Dump می‌کنید یا در یک فایل در جایی ذخیره Store می‌کنید.

LOAD: مثل ویژگی‌های هدوپ، اشیائی که توسط هدوپ ذخیره می‌شوند در HDFS ذخیره می‌شوند. به منظور اینکه برنامه Pig به این داده‌ها دسترسی پیدا کند، برنامه باید اول به Pig بگوید چه فایل (یا فایل‌هایی) استفاده خواهد کرد و توسط دستور Load 'data-file' اجرا خواهد شد. (که 'data-file' می‌تواند یک فایل یا یک دایرکتوری HDFS باشد). اگر دایرکتوری شناسایی شود کل فایل‌های آن به برنامه بار خواهند شد. اگر داده‌های بار شده در قالب قابل شناسایی توسط Pig نباشند می‌توانید توابع using را استفاده کنید تا ازآن استفاده کنند.

TRANSFORM: منطق transformبه همه داده‌های دستکاری شده اعمال می‌شود. در اینجا شما می‌توانید سطرهایی که نمی‌خواهید را FILTER کنید، دو مجموعه داده را JOIN کنید، داده‌ها را GROUP کنید و نتایج را مرتب کنید و ... در زیر مثالی از برنامه Pig آورده شده است که فایل‌های ترکیبی از Twitter را می‌گیرد، توییت‌هایی را انتخاب می‌کند که iso-language code آنها en (English) است، سپس آنها را بر اساس کاربرانی که توییت کرده‌اند گروه بندی می‌کند و مجموع تعداد توییت‌های مجدد را می‌شمارد.

```
L  = LOAD 'hdfs//node/tweet_data';
FL = FILTER L BY iso_language_code  EQ 'en';
G  = GROUP FL BY from_user;
RT = FOREACH G GENERATE group, SUM(retweets);
```

DUMP و STORE: اگر شما دو دستور DUMP و STORE را مشخص نکنید، نتایج برنامه Pig تولید نمی‌شوند. شما معمولاً از دستور DUMP استفاده می‌کنید تا نتایج را در صفحه نمایش نشان دهد. برای ذخیره نتایج هم می‌توانید از دستور STORE برای آنالیز بیشتر استفاده کنید. نکته اینکه شما می‌توانید دستور DUMP در هر جایی از برنامه‌تان استفاده کنید تا مجموعه نتایج میانی را در صفحه نشان دهد، که برای اهداف دیباگ بسیار سودمند است.

نکته اینکه ما یک برنامه Pig داریم که برای اجرا باید از محیط هدوپ استفاده کرد. سه راه برای اجرای یک برنامه Pig وجود دارد: تعبیه در یک اسکریپت، تعبیه در یک برنامه جاوا، یا اجرا از خط فرمان Pig به نام Grunt.

مهم نیست برنامه‌تان را از چه روشی اجرا می‌کنید، در زمان اجرا محیط Pig برنامه را به مجموعه‌ای از کارهای map و reduce ترجمه می‌کند و آن را اجرا می‌کند. این روش آنالیز مقدار زیاد داده را ساده می‌کند و به توسعه دهندگان اجازه می‌دهد تا روی آنالیز داده تمرکز کنند نه به کارهای map و reduce.

Hive: هرچند Pig یک زبان ساده و قوی است اما جدید است و نیاز به آموزش دارد. بعضی افراد در فیس بوک یک ساختار پشتیبانی هدوپ ساختند که به هر فردی که با SQL آشنا است (که برای توسعه دهندگان پایگاه داده رابطه ای رایج می‌باشد) بتواند پلتفرم هدوپ را استفاده کند. اختراعشان Hive نامیده می‌شود که به توسعه دهندگان SQL اجازه می‌دهد از دستورات زبان پرس و جوی Hive یعنی HQL استفاده کنند که مشابه دستورات SQL استاندارد است. باید بدانید که HQL به دستوراتی که می‌فهمد محدود است. اما استفاده از آن جالب است. دستورات HQL توسط Hive به وظیفه‌های MapReduce شکسته می‌شوند و در خوشه هدوپ اجرا می‌شوند.

برای هرکسی که با پایگاه رابطه ای SQL آشنا است، این بخش خیلی آشنا خواهد بود. زیرا با هر سیستم مدیریت پایگاه داده DBMS می‌توان پرس و جوهای Hive را به روش‌های مختلف اجرا کرد. شما می‌توانید آنها را از خط فرمان Hive اجرا کنید، یا از اتصال پایگاه داده جاوا (JDBC)، یا اتصال پایگاه داده باز ODBC یا از طریق Hive Thrift Client. مثال زیر روش ایجاد یک جدول، دستکاری آن، و پرس و جوی آن را با استفاده از Hive نشان می‌دهد:

```
CREATE TABLE Tweets(from_user STRING, userid BIGINT, tweettext STRING,
    retweets INT)
 COMMENT 'This is the Twitter feed table'
 STORED AS SEQUENCEFILE;
LOAD DATA INPATH 'hdfs://node/tweetdata' INTO TABLE TWEETS;
SELECT from_user, SUM(retweets)
 FROM TWEETS
 GROUP BY from_user;
```

همانطور که می‌بینید، Hive شبیه کد پایگاه داده SQL سنتی می‌باشد. به هر حال چون براساس عملیات MapReduce و هدوپ می‌باشد، چندین تفاوت کلیدی وجود دارد: اول اینکه هدوپ برای پیمایش‌های ترتیبی طولانی می‌باشد و چون Hive براساس هدوپ است، تاخیر آن زیاد است. یعنی Hive مناسب کاربردهایی که نیازمند زمان‌های پاسخ خیلی سریع هستند نیست. در انتها Hive، براساس خواندن است و برای پردازش‌های تراکنشی مناسب نیست که عموماً شامل درصد بالایی از عملیات نوشتن هستند.

مثالی از Job با Mapreduce:

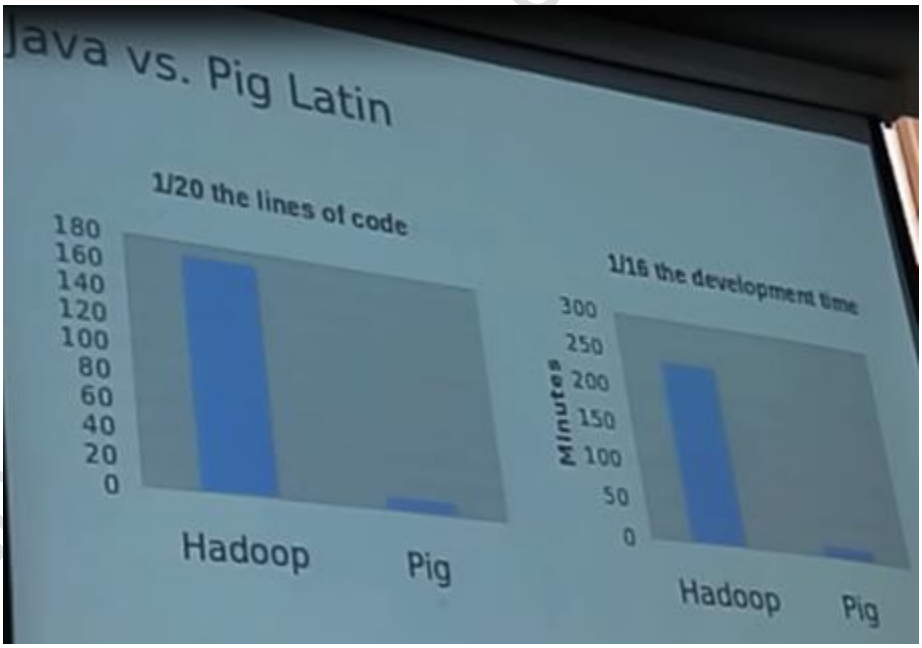
فرض می‌کنیم میزان برق مصرفی یک سازمان را داریم و می خواهیم با MapReduce میانگین مصرف، کمترین مصرف و بیشترین مصرف را بدست آوریم. کدی که با MapReduce باید بنویسیم به صورت زیر است:

```
package hadoop;
import java.util.*;
import java.io.IOException;
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
```

```
public class ProcessUnits
{
   //Mapper class
   public static class E_EMapper extends MapReduceBase implements
   Mapper<LongWritable ,/*Input key Type */
   Text,                /*Input value Type*/
   Text,                /*Output key Type*/
   IntWritable>        /*Output value Type*/
   {

      //Map function
      public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException
      {
         String line = value.toString();
         String lasttoken = null;
         StringTokenizer s = new StringTokenizer(line,"\t");
         String year = s.nextToken();

         while(s.hasMoreTokens())
            {
               lasttoken=s.nextToken();
            }

         int avgprice = Integer.parseInt(lasttoken);
         output.collect(new Text(year), new IntWritable(avgprice));
      }
   }


   //Reducer class
   public static class E_EReduce extends MapReduceBase implements
   Reducer< Text, IntWritable, Text, IntWritable >
   {

      //Reduce function
      public void reduce( Text key, Iterator <IntWritable> values,
     OutputCollector<Text, IntWritable> output, Reporter reporter) throws
IOException
         {
            int maxavg=30;
            int val=Integer.MIN_VALUE;

            while (values.hasNext())
            {
               if((val=values.next().get())>maxavg)
               {
                  output.collect(key, new IntWritable(val));
               }
            }

         }
   }
```

```
    //Main function
    public static void main(String args[])throws Exception
    {
       JobConf conf = new JobConf(ProcessUnits.class);

       conf.setJobName("max_eletricityunits");
       conf.setOutputKeyClass(Text.class);
       conf.setOutputValueClass(IntWritable.class);
       conf.setMapperClass(E_EMapper.class);
       conf.setCombinerClass(E_EReduce.class);
       conf.setReducerClass(E_EReduce.class);
       conf.setInputFormat(TextInputFormat.class);
       conf.setOutputFormat(TextOutputFormat.class);

       FileInputFormat.setInputPaths(conf, new Path(args[0]));
       FileOutputFormat.setOutputPath(conf, new Path(args[1]));

       JobClient.runJob(conf);
    }
}
```

مقایسه Pigو Hadoop MapReduce: همانطور که گفتیم کار با MR نیازمند برنامه نویسی جاوا و مهارت در آن می‌باشد. ولی Pig کارها را ساده‌تر می‌کند و در انتها به Job های MR تبدیل می‌شوند. همانطور که در نمودار شکل ۳- ۱۱ مشاهده می‌کنید هم از نظر تعداد خطوط کد و هم سرعت اجرا Pig بهتر می‌باشد.



شکل ۳- ۱۱: مقایسه Javaبا PigLatin

Google BigTable-۲

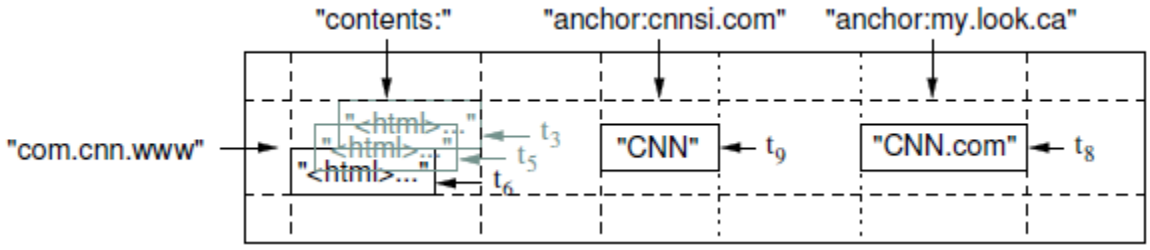
BigTable یک سیستم ذخیره سازی توزیع شده برای مدیریت داده‌های ساختار یافته می‌باشد که برای مقیاس‌پذیر شدن تا اندازه‌های بسیار بزرگ می‌باشد: یعنی چندین پتابایت داده در هزاران سرور. پروژه‌های زیادی در گوگل داده‌ها را در BigTable ذخیره می‌کنند از ایندکس کردن وب تا Google Earth و کارهای مالی گوگل. این کاربردها از BigTable درخواست های متعددی دارند هم از لحاظ اندازه داده‌ها (از URL های صفحات وب تا تصاویر ماهواره‌ای) و هم میزان تاخیر (Latency) (بعضی نیازمند پردازش میزان داده حجیم هستند و بعضی نیازمند سرویس بلادرنگ می‌باشند). اما بدون توجه به این نیازهای متفاوت BigTable راه‌حل موفق، منعطف و با کارایی بالا برای همه محصولات گوگل فراهم کرده است.

BigTable به چندین هدف رسیده است: کاربرد گسترده، مقیاس‌پذیری، کارایی بالا، و قابل دسترس بودن بالا. BigTable توسط بیشتر از ۶۰ محصول و پروژه گوگل استفاده می‌شود مثل Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. این محصولات BigTable را برای بارهای کاری مختلفی استفاده می‌کنند که از کارهای پردازشی دسته‌ای نیازمند Throughput تا سرویس‌های نیازمند پاسخ بلادرنگ برای کاربران متغیرند. خوشه‌های BigTable استفاده شده توسط این محصولات پیکربندی‌های متفاوتی دارند، از چند تا هزاران سرور و تا چند صد ترابایت داده را ذخیره می‌کنند.

از خیلی جنبه‌ها BigTable مشابه یک پایگاه داده می‌باشد: راهبردهای پیاده‌سازی زیادی با پایگاه داده دارد. پایگاه داده‌های موازی و پایگاه داده‌های حافظه اصلی به کارایی و مقیاس‌پذیری بالا می‌رسند اما BigTable رابطهای متفاوتی از چنین سیستم‌های دارد. BigTable یک مدل پایگاه داده رابطه‌ای کامل را پشتیبانی نمی‌کند، اما برای کلاینت‌ها کنترل پویا روی لایه‌های داده و شکل آنها فراهم می‌کند. داده‌ها با نام‌های سطر و ستون ایندکس می‌شوند که می‌توانند رشته‌های با طول دلخواه باشند. کلاینت‌ها اغلب داده‌های ساختار یافته و شبه ساختار یافته را در این رشته ها قرار می‌دهند. کلاینت‌ها می‌توانند روی محلیت داده‌هایشان کنترل داشته باشند. همچنین می‌توانند تعیین کنند چه موقع داده‌ها در حافظه یا دیسک قرار گیرند.

مدل داده BigTable: BigTable یک map خلوت (Sparse)، توزیع شده، دائما مرتب شده چند بُعدی می‌باشد. این map با یک کلید سطر، کلید ستون و یک مهر زمانی ایندکس می‌شود. هر مقدار در map آرایه‌ای تفسیر نشده از بایت‌ها می‌باشد.

(row:string, column:string, time:int64) → string



شکل ۳- ۱۲: بخشی از یک جدول نمونه که صفحات وب را ذخیره می‌کند. نام سطر URL معکوس شده است. ستون خانواده محتوا (content Column family) شامل محتوای صفحه، ستون خانواده anchor شامل متن هر anchorی می‌باشد که به صفحه رجوع می‌دهد. صفحه وب CNN هم توسط صفحات وب Sport Illustrated و هم My-look رجوع داده شده است بنابراین شامل ستون‌های با نام anchor:cnnsi.com و anchor:my.look.ca می‌باشد. هر سلول anchor یک نسخه دارد، ستون محتوا سه نسخه در مهرهای زمانی t3,t5,t6 دارد.

در شکل ۳- ۱۲ یک نمونه از BigTable نوعی برای ذخیره صفحات وب را نشان می‌دهد. که در اینجا آن را Webtable می‌نامیم.

سطرها (Rows) :کلیدهای سطر در جدول رشته‌های دلخواه هستند (که اکنون اندازه آنها تا 64KB می‌باشد، هرچند که اکثر کاربران ۱۰ تا ۱۰۰ بایت را استفاده می‌کنند). هر خواندن و نوشتن در یک سطر کلید بصورت اتمیک انجام می‌شود (بدون توجه به تعداد ستون‌های متفاوتی که در یک سطر خوانده یا نوشته می‌شوند).

BigTable داده‌ها را براساس کلید سطر مرتب (lexicographic) نگه می‌دارد و باعث کنترل محلیت می‌شود یعنی سطرهای مشابه در مکان‌های متوالی قرار می‌گیرند. محدوده سطر بطور پویا قسمت بندی می‌شود. هر محدوده سطر یک tablet نامیده می‌شود که واحد توزیع و تعادل بار می‌باشد. در نتیجه خواندن محدوده کوچکی از سطرها کارامد می‌باشد و نیازمند ارتباط با تعداد کمی ماشین می‌باشد. کلاینت‌ها می‌توانند از این خصوصیت بهره‌برداری کنند بطوریکه محلیت خوبی برای دسترسی داده‌هایشان ایجاد کنند. به عنوان مثال در Webtable صفحات در دامنه مشابه با هم در سطرهای متوالی با معکوس کردن نام میزبان (hostname) URL گروه بندی می‌شوند. به عنوان مثال ما داده‌های maps.google.com/index.html را با کلید com.google.maps/index.html ذخیره می‌کنیم. ذخیره صفحات دامنه‌های مشابه در کنار هم آنالیز صفحات و میزبان‌های دامنه را کارامدتر می‌کند.

خانواده‌های ستون Column Families: کلیدهای ستون در مجموعه‌هایی به نام خانواده ستون گروه‌بندی می‌شود، که واحد اصلی کنترل دسترسی را شکل می‌دهد. همه داده‌های ذخیره شده در یک خانواده ستون معمولاً از نوع مشابهی هستند (ما داده‌های از نوع مشابه را با هم در یک خانواده ستون می‌فشاریم(compress)). قبل از اینکه داده‌ها تحت هر کلید ستون ذخیره شوند باید خانواده ستون ایجاد شود، بعد از اینکه یک خانواده ایجاد شد، هر کلید ستون در داخل خانواده می‌تواند استفاده شود. ما دوست داریم که تعداد خانواده‌های ستون مجزا در هر جدول کم باشد (حداکثر صدها عدد)، و آن خانواده‌ها به ندرت در طول عملیات تغییر کنند. در مقایسه، تعداد ستون‌ها در جدول می‌تواند بدون کران باشد. (unbounded).

یک کلید ستون با استفاده از ساختار زیر نام‌گذاری می‌شود: family:qualifier. نام‌های خانواده ستون باید قابل پرینت باشد، اما qualifier ها ممکن است رشته های دلخواه باشند. به عنوان مثال در Webtable یک خانواده ستون می‌تواند language باشد، که ID زبان‌های صفحات وب را ذخیره کند. یک خانواده ستون دیگر برای این جدول anchor می‌باشد که هر کلید ستون در این خانواده یک anchor است که در شکل۳– ۱۲ نشان داده شده است. Qualifier نام سایت مراجعه کننده (referring) می‌باشد، محتوای سلول متن لینک است.

کنترل دسترسی و memory accounting در سطح خانواده–ستون انجام می‌شود. در Webtable مثال ما، این کنترل‌ها به ما اجازه می‌دهد کاربردهای متنوعی را مدیریت کنیم: بعضی داده‌های اصلی را اضافه می‌کنند، بعضی داده‌های اصلی را می‌خوانند و خانواده‌های ستون را استخراج می‌کنند، و بعضی فقط مجازند داده‌های موجود را مشاهده کنند (حتی برای امنیت به همه خانواده‌های موجود دسترسی ندارند).

Timestamps: هر سلول در BigTable می‌تواند چندین نسخه از یک داده را داشته باشد، این نسخه‌ها با مهر زمانی ایندکس می‌شوند. در Bigtable timestamp اعداد صحیح ۶۴ بیتی می‌باشند. آنها می‌توانند توسط BigTable مقدار دهی شوند، که در این حالت می‌توانند زمان واقعی را براساس میکروثانیه نشان دهند، یا بطور صریح توسط کاربردهای کلاینت مقدار دهی شوند. کاربردهایی که می‌خواهند از تصادم جلوگیری کنند باید خودشان timestampهای منحصر بفرد تولید کنند. نسخه‌های مختلف یک سلول به ترتیب نزولی مهرزمانی ذخیره می‌شوند، بطوریکه اخیرترین نسخه بتواند در ابتدا خوانده شود.

برای اینکه مدیریت نسخه‌های داده خیلی سنگین نشود، ما از دو تنظیم برای هر خانواده ستون پشتیبانی می‌کنیم که به Bigtable می‌گوید بطور خودکار نسخه‌های سلول را به سطل آشغال بریزد–یا جمع‌آوری کند (garbage-collect). کلاینت می‌تواند تعیین کند که فقط نسخه‌های به اندازه کافی جدید نگه داشته شوند یا n نسخه اخیر سلول حفظ شوند. (مثلا، فقط مقادیری حفظ شوند که در ۷ روز اخیر نوشته شده‌اند).

API: BigTable API توابعی برای ایجاد و حذف جداول و خانواده ستون‌ها دارد. همچنین توابعی برای تغییر خوشه، جدول، و متادیتای خانواده ستون مثل حقوق کنترل دسترسی دارد.

کاربردهای کلاینت می‌توانند مقادیر Bigtable را نوشته یا حذف کنند، مقادیر را از سطرهای مجزا جستجو کنند، یا در زیر مجموعه ای از سطرهای داده در جدول بچرخند. شکل ۳-۱۳ کد C++ی را نشان می‌دهد که RowMutation abstraction را برای انجام یک سری بروزرسانی استفاده می‌کند. (جزئیات نامربوط حذف شده‌اند تا مثال کوتاه شود). Apply یک عمل اتمیک در Webtable انجام می‌دهد: آن یک انکر را به www.cnn.com اضافه می‌کنند و یک انکر دیگر را حذف می‌کند.

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

شکل ۳-۱۳: نوشتن در Bigtable

شکل ۳-۱۴ کد C++ی را نشان می‌دهد که Scanner abstraction را استفاده می‌کند تا همه انکرها را در یک سطر خاص ببینند. کلاینت‌ها می‌توانند روی چندین خانواده ستون مختلف بچرخند و چندین مکانیزم برای محدود کردن سطرها، ستون‌ها، و مهرهای زمانی تولید شده با یک scan وجود دارند. به عنوان مثال، می‌توانیم scan فوق را طوری محدود کنیم که فقط انکرهایی را بسازد که ستون‌ها با عبارت منظم anchor:*cnn.com تطابق داشته باشد یا فقط انکرهایی را تولید کند که مهرهای زمانی آن در ده روز از زمان جاری قرار بگیرند.

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

شکل ۳-۱۴: خواندن از Bigtable

BigTable چندین ویژگی دیگر را پشتیبانی می‌کند که به کاربر اجازه می‌دهد داده‌ها را به روش‌های پیچیده‌تری دستکاری کند. اولاً، Bigtable همه تراکنش‌های تک-سطر را پشتیبانی می‌کند، که می‌تواند برای انجام رشته read-modify-write اتمیک روی داده‌های ذخیره شده در یک کلید سطر استفاده شود. ثانیاً، bigtable به سلول‌ها اجازه می‌دهد تا به عنوان شمارنده‌های صحیح استفاده شوند و در آخر، Bigtable اجرای اسکریپت‌های تهیه شده توسط کلاینت در فضای آدرس سرورها را پشتیبانی می‌کند. اسکریپت‌ها در زبانی نوشته می‌شوند که توسط گوگل برای پردازش داده‌ها به نام Sawzall توسعه داده شده است.

Bigtable می‌تواند با MapReduce استفاده شود، یک قطعه کاری برای اجرای محاسبات موازی در مقیاس بزرگ توسعه داده شده در گوگل. گوگل مجموعه ای wrapper نوشته است که به bigtable اجازه می‌دهد هم به عنوان منبع ورودی و هم مقصد خروجی MapReduce Jobs استفاده شود.

Building Blocks: Bigtable روی قطعات دیگر زیرساخت گوگل ساخته شده است. Bigtable سیستم فایل توزیع شده گوگل (GFS[1]) را استفاده می‌کند تا فایل‌های داده و log را ذخیره کند. یک خوشه Bigtable معمولاً در یک استخر مشترک از ماشین‌ها عمل می‌کند که انواع گوناگونی از سایر کاربردهای توزیع شده را اجرا می‌کنند، و Bigtable معمولاً همان ماشینی را به اشتراک استفاده می‌کند که پردازه‌های سایر کاربردها را دارد. Bigtable وابسته به یک سیستم مدیریت خوشه برای زمانبندی jobها، مدیریت منابع در ماشین‌های مشترک، کار با خرابی (شکست) های ماشین، و مانیتور کردن وضعیت ماشین می‌باشد.

قالب فایل SSTable بطور داخلی برای ذخیره داده‌های bigtable استفاده می‌شود. SSTable یک immutable map مرتب شده دائمی از کلیدها به مقادیر فراهم می‌کند، که هر دوی کلیدها و مقادیر رشته بایت‌های دلخواه هستند. عملیاتی برای جستجوی مقادیر مرتبط با یک کلید تعیین شده، و چرخیدن در همه زوج key/value ها در یک دامنه کلید تعیین شده تهیه شده اند. بطور داخلی هر SSTable شامل یک رشته بلوک می‌باشد (که معمولاً اندازه 64KB دارند، اما قابل پیکربندی می‌باشند). ایندکس بلوک (که معمولاً در انتهای SSTable ذخیره می‌شود) برای تشخیص مکان بلوک‌ها استفاده می‌شود، ایندکس وقتیکه SSTable باز می‌شود به حافظه (memory) بار می‌شود. یک جستجو معمولاً با یک دسترسی (disk seek) به دیسک انجام می‌شود: ما ابتدا بلوک مناسب را با انجام جستجوی باینری (binary search) در ایندکس داخل حافظه (in-memory) پیدا می‌کنیم، و سپس بلوک مناسب را از دیسک می‌خوانیم. معمولاً SSTable می‌تواند بطور کامل در داخل حافظه RAM نگاشت شود، که به ما اجازه می‌دهد جستجو و اسکن را بدون لمس دیسک انجام دهیم.

Bigtable روی یک سرویس قفل توزیع شده دائمی و با دسترس پذیری بالا به نام chubby تکیه دارد. یک سرویس Chubby شامل پنج کپی (replica) فعال می‌باشد که یکی از آنها به عنوان ارباب (master) انتخاب می‌شود که بطور فعال به درخواست ها پاسخ می‌دهد. سرویس تا وقتیکه اکثریت کپی‌ها در حال اجرا باشند و بتوانند با یکدیگر ارتباط برقرار کنند زنده است. Chubby از الگوریتم Paxos استفاده می‌کند تا کپی‌هایش را در مواجهه با شکست‌ها consistent نگه دارد. Chubby فضای نامی را تهیه می‌کند که شامل فایل‌ها و دایرکتوری‌های کوچک است. هر دایرکتوری یا فایل می‌تواند به عنوان یک قفل استفاده شود، و خواندن نوشتن‌ها در یک فایل اتمیک می‌باشد. کتابخانه کلاینت Chubby کش کردن consistent فایل‌های Chubby را فراهم می‌کند. هر کلاینت Chubby یک جلسه[2] با یک سرویس Chubby نگه می‌دارد. جلسه کلاینت منقضی خواهد شد اگر نتواند در دوره انقضای جلسه آن را تازه‌سازی کند. وقتی جلسه یک کلاینت منقضی شود، همه قفل‌ها و موارد مرتبط را از دست خواهد داد.

Bigtable از Chubby برای کارهای مختلفی استفاده می‌کند: تا اطمینان دهد که حداکثر یک ارباب فعال در هر لحظه وجود دارد، تا مکان bootstrap داده‌های bigtable را ذخیره کند، تا سرور tablet را کشف کند و اطلاعات شمای bigtable (اطلاعات خانواده ستون براس هر جدول) را ذخیره کند و تا لیست های کنترل دسترسی را ذخیره کند. اگر chubby در یک دوره زمانی توسعه غیرقابل دسترس شود، bigtable غیرقابل دسترس خواهد شد. در یک آزمایش این اثر روی ۱۴ خوشه Bigtable که ۱۱ نمونه chubby را استفاده می‌کردند بررسی شده است. درصد میانگینی که بعضی داده‌های ذخیره شده در Bigtable به خاطر غیرقابل دسترس بودن Chubby، غیرقابل دسترس

[1] Google File System
[2] Session

بوده‌اند 0.0047% بوده است. درصدی که یک خوشه به خاطر غیرقابل دسترس بودن Chubby تحت تاثیر قرار گرفته است 0.0326% بوده است.

پیاده سازی Implementation:پیاده‌سازی Bigtable سه جزء اصلی دارد: یک کتابخانه که به هر کلاینت لینک می‌شود، یک سرور ارباب، و تعداد زیادی سرور Tablet. سرورهای Tablet می‌توانند بطور پویا به یک خوشه اضافه (یا حذف) شوند تا با تغییرات بارهای کاری منطبق شوند.

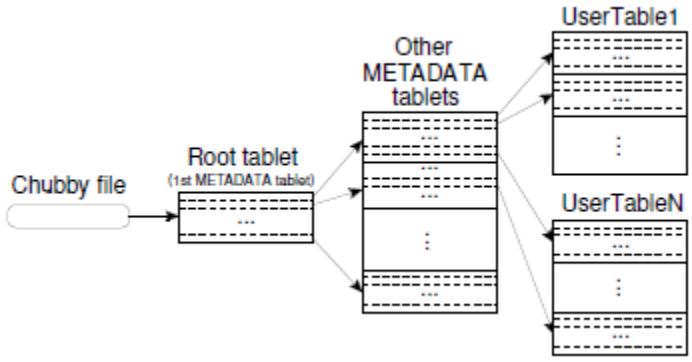
ارباب مسئول اختصاص دادن tabletها به سرورهای tablet، تشخیص اضافه شدن و منقضی شدن سرورهای Tablet، متعادل کردن بار tablet-server، و جمع‌آوری–آشغال کردن فایل‌های GFS می‌باشد. به علاوه تغییرات شماتیکی مثل ایجاد جدول و خانواده‌های ستون را مدیریت می‌کند.

هر سرور tablet یک مجموعه tablet دارد (معمولاً هر سرور تبلت بین ده تا هزار تبلت دارد). سرور تبلت درخواست‌های خواندن و نوشتن به تبلت‌هایی که بارگذاری کرده است را مدیریت می‌کند و معمولاً تبلت‌هایی که بیش از حد بزرگ شده‌اند را به چند تبلت تقسیم می‌کند.

مثل اکثر سیستم‌های ذخیره‌سازی توزیع شده با یک ارباب، داده‌های کلاینت از طریق ارباب منتقل نمی‌شوند: کلاینت‌ها برای خواندن و نوشتن مستقیماً با سرورهای تبلت ارتباط برقرار می‌کنند. چون کلاینت‌های Bigtable روی ارباب برای اطلاعات مکان تبلت تکیه ندارند، اکثر کلاینت‌ها هیچ‌گاه با ارباب ارتباط برقرار نمی‌کنند. در نتیجه عملاً بار ارباب سبک می‌باشد.

یک خوشه Bigtable تعدادی جدول ذخیره می‌کند. هر جدول شامل یک مجموعه تبلت می‌باشد و هر تبلت شامل همه داده‌های مربوط به یک دامنه داده می‌باشد. در ابتدا، هر جدول فقط شامل یک تبلت می‌باشد. وقتی یک جدول رشد می‌کند، بطور خودکار به چندین تبلت شکسته می‌شود، که هر کدام تقریبا بطور پیش‌فرض اندازه ۱۰۰ تا ۲۰۰ مگابایت دارند.

مکان تبلت Tablet Location:برای ذخیره اطلاعات مکان تبلت سلسله مراتب سه–سطحه مشابه با B^+tree استفاده می‌شود. (شکل ۳–۱۵)



شکل ۳– ۱۵: سلسله مراتب مکان تبلت

سطح اول یک فایل است که در chubby ذخیره می‌شود و شامل مکان تبلت ریشه است. تبلت ریشه شامل مکان همه تبلت‌ها در یک جدول متادیتای خاص می‌باشد. هر تبلت متادیتا شامل مکان یک مجموعه از تبلت‌های کاربر می‌باشد. تبلت ریشه اولین تبلت در جدول متادیتا می‌باشدف اما با آن بطور ویژه رفتار می‌شود، هیچگاه به چند بخش تقسیم نمی‌شود– تا اطمینان بدهد که سلسله مراتب مکان تبلت بیشتر از سه سطح ندارد.

جدول متادیتا مکان یک تبلت را به یک کلید سطر نگه می‌دارد که رمزگذاری شده شناسه جدول تبلت و سطر آخر آن می‌باشد. هر سطر متادیتا تقریباً 1kB داده در حافظه ذخیره می‌کند. با میانگین تبلت متادیتای 128MBی، شماتیک سلسله مراتبی سه سطحی برای آدرس دادن 2^{34} تبلت مناسب است (یا 2^{61} بایت در تبلت‌های 128MBی).

کتابخانه کلاینت مکان تبلت را کش می-کند. اگر کلاینت مکان یک تبلت را نداند، یا اگر کشف کند که اطلاعات مکان کش شده درست نیستند، بطور بازگشتی در سلسله مراتب مکان تبلت حرکت می‌کند. اگر کش کلاینت خالی باشد، الگوریتم مکان نیازمند سه زمان چرخه[1] شبکه شامل یک خواندن از Chubby می‌باشد. اگر کش کلاینت کهنه باشد، الگوریتم مکان تا شش زمان چرخه طول خواهد کشید، زیرا ورودی‌های کهنه مش فقط در فقدان‌ها[2] کشف می‌شوند (با این فرض که تبلت‌های متادیتا بطور متناوب جابجا نمی‌شوند). چون مکان‌های تبلت‌ها در حافظه RAM ذخیره می‌شوند، نیازی به دسترسی به GFS نیست.

ما همچنین اطلاعات ثانویه را در جدول متادیتای تبلت ذخیره می‌کنیم، شامل log همه رخدادهای مرتبط با هر تبلت (مثل چه موقع یک سرور به آن سرویس می‌دهد). این اطلاعات مفید برای دیباگ کردن و آنالیز کارایی مفید هستند.

تخصیص تبلت Tablet Assignment: هر تبلت به یک سرور تبلت در یک زمان تخصیص داده می‌شود. ارباب مجموعه سرورهای تبلت زنده را ردیابی می کند و همچنینی تخصیص‌های جاری تبلت ها به سرورهای تبلت شامل تبلت‌هایی که تخصیص نیافته اند. وقتیکه یک تبلت تخصیص داده نشده باشد، و سرور تبلت با فضای کافی برای تبلت در دسترس باشد، ارباب تبلت را با فرستادن درخواست بارگذاری تبلت به سرور تبلت تخصیص می‌دهد.

Bigtable از Chubby استفاده می‌کند تا سرورهای تبلت را ردیابی کند. وقتیکه یک سرور تبلت شروع می‌کند، یک قفل انحصاری را ایجاد و دریافت می‌کند که یک فایل با نام منحصر بفرد در یک دایرکتوری Chubby می‌باشد. ارباب این فهرست را مانیتور می‌کند (دایرکتوری‌های سرورها) تا سرورهای تبلت را کشف کند. یک سرور تبلت از سرویس دادن به تبلت‌هایش متوقف می‌شود اگر قفل منحصر بفردش را گم کند: مثلاً به خاطر خطاهای شبکه. (Chubby یک مکانیزم کارامد دارد که به یک سرور تبلت اجازه می‌دهد بدون ایجاد ترافیک در شبکه چک کند قفلش را هنوز باید نگه دارد). یک سرور تبلت تلاش خواهد کرد تا یک قفل منحصر بفرد روی فایل‌هایش بدست آورد تا زمانیکه هنوز فایلی دارد. اگر فایلی وجود نداشته باشد آنگاه سرور تبلت قادر به سرویس دهی نخواهد بود و خودکشی خواهد کرد. وقتیکه کار یک سرور تبلت تمام شد (مثلاً، سیستم مدیر خوشه ماشین سرور تبلت را از خوشه حذف کند)، تلاش خواهد کرد تا قفلش را آزاد کند بطوریکه ارباب دوباره تبلت‌هایش را سریعتر تخصیص دهد.

ارباب مسئول این است که تشخیص دهد چه موقع یک سرور تبلت به تبلت‌هایش سرویس نمی‌دهد، و آن تبلت‌ها را هرچه زودتر مجدد تخصیص دهد. برای تشخیص اینکه چه موقع یک سرور تبلت به تبلت‌هایش سرویس نمی‌دهد از سرورهای تبلت بطور دوره‌ای راجع به وضعیت قفل‌شان می‌پرسد. اگر یک سرور تبلت گزارش بدهد که قفلش را گم کرده است، یا اگر ارباب قادر به رسیدن به یک سرور در چند تلاش اخیرش نباشد، ارباب تلاش می‌کند تا یک قفل منحصر بفر در سرور فایل بدست آورد. اگر ارباب قادر یه بدست آوردن قفل بود، آنگاه Chubby زنده است و سرور تبلت چه مرده باشد یا برای رسیدن به Chubby به مشکل برخورده باشد، ارباب اطمینان می‌دهد که آن سرور تبلت هیچگاه نمی‌تواند سرویس بدهد با حذف فایل سرور آن. وقتیکه یک فایل سرور حذف شد، ارباب می‌تواند همه تبلت‌هایی که قبلاً به آن سرور تخصیص داده بود را به مجموعه ای از تبلت‌های تخصیص نیافته انتقال دهد. برای اطمینان از اینکه خوشه Bigtable به خاطر مسائل شبکه بین ارباب و Chubby

[1] Round-trip
[2] Misses

آسیب‌پذیر نیست، ارباب در صورت منقضی شدن Chubby خودکشی خواهد کرد. به هر حال همانطور که در بالا گفتیم شکست ارباب تخصیص تبلت‌ها به سرورهای تبلت را تغییر نمی‌دهد.

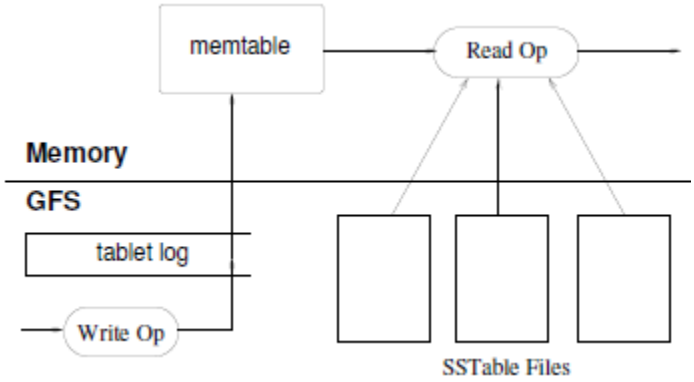
وقتیکه یک ارباب توسط سیتم مدیریت خوشه شروع می‌کند، نیاز دارد تا تخصیص‌های جاری تبلت‌ها را کشف کند قبل از اینکه بتواند آنها را تغییر دهد. ارباب مراحل زیر را در موقع شروع اجرا می‌کند: ۱) ارباب یک قفل ارباب منحصر بفرد در Chubby بدست می‌آورد، که از شروع همروند چند ارباب جلوگیری می‌کند. ۲) ارباب دایرکتوری سرورها را در Chubby اسکن می‌کند تا سرورهای زنده را پیدا کند. ۳) ارباب با هر سرور تبلت زنده ارتباط برقرار می‌کند تا تبلت‌هایی که هم اکنون به هر سرور تخصیص داده شده است را کشف کند. ۴) ارباب جدول متا دیتا را اسکن می‌کند تا مجموعه تبلت‌ها را یاد بگیرد. هرگاه این اسکن با تبلتی برخورد کرد که هم اکنون تخصیص نیافته است، ارباب تبلت را به مجموعه تبلت های تخصیص نیافته اضافه می‌کند، که باعث می‌شود تبلت برای تخصیص واجد شرایط شود.

یک پیچیدگی این است که اسکن جدول متادیتا تا زمانیکه تبلت‌های متادیتا تخصیص داده نشده باشند نمی‌تواند انجام شود. بنابراین، قبل از شروع این اسکن (مرحله ۴)، ارباب تبلت ریشه را به مجموعه تبلت‌های تخصیص نیافته اضافه می‌کند اگر انتسابی برای تبلت ریشه در طول مرحله ۳ انجام نشده باشد. این اضافه کردن تضمین می‌دهد که تبلت ریشه باید انتساب داده شود. چون تبلت ریشه دربرگیرنده نام‌های همه تبلت‌های متادیتا می‌باشد، ارباب همه آنها را بعد از اسکن تبلت ریشه می‌داند.

مجموعه تبلت‌های موجود فقط زمانی تغییر می‌کند که، یک تبلت ایجاد یا حذف شود، دو تبلت موجود با هم ادغام شوند و تبلت بزرگتری را شکل دهند، یا یک تبلت موجود به دو تبلت کوچکتر تقسیم شود. ارباب بجز آخرین گزینه قادر به ردیابی همه این تغییرات هست. گزینه تقسیم تبلت به روش متفاوتی انجام می‌شود، چون توسط یک سرور تبلت آغاز مش‌شود. سرور تبلت تقسیم را با ضبط اطلاعاتی برای تبلت جدید در جدول متادیتا انجام می‌دهد. وقتیکه تقسیم تمام شد، به ارباب خبر می‌دهد. در حالتیکه پیام تقسیم تبلت گم شود (به خاطر مرگ سرور تبلت یا ارباب) ، ارباب تبلت جدید را هنگامیکه از یک تبلت سرور بخواهد تبلت جدیدی که هم اکنون تقسیم شده است را بارگذاری کند، تشخیص می‌دهد. سرور تبلت به ارباب راجع به تقسیم خواهد گفت.

سرویس دهی به تبلت Tablet Serving: همانطور که در شکل ۳– ۱۶ مشاهده می‌کنید، حالت دائمی یک تبلت در GFS ذخیره می‌شود. بروزرسانی‌ها در یک commit log انجام می‌شوند که رکوردهای redo را ذخیره می‌کند. از این بروزرسانی‌ها، بروزرسانی‌هایی که اخیراً انجام شده‌اند در حافظه در یک بافر مرتب شده به نام memTable ذخیره می‌شوند، بروزرسانی‌های قدیمی‌تر در رشته‌ای از SSTableها ذخیره می‌شوند. برای بازیابی یک تبلت، یک سرور تبلت متادیتای آن را از جدول متادیتا می‌خواند. این متادیتا شامل لیستی از SSTableها می‌باشد که دربرگیرنده یک تبلت هستند و یک مجموعه از نقاط redo، که اشاره‌گرهایی به هر commit log هستند که ممکن است شامل داده‌هایی برای تبلت باشند. سرور اندیس‌های SSTable را به داخل حافظه (memory) می‌خواند و memTable را با اعمال همه بروزرسانی‌هایی که تمام شده‌اند بازسازی می‌کند.

هنگامیکه یک عمل نوشتن به یک سرور تبلت می‌رسد، سرور چک می‌کند ببیند Well-formed هست و فرستنده برا انجام تغییر مجاز هست!. اعتبارسنجی با خواندن لیستی از نویسنده‌های مجاز از یک فایل Chubby (که تقریباً همیشه در کش کلاینت chubby موجود هست) انجام می‌شود. یک تغییر معتبر در commit log نوشته می‌شود. Group commit برای بهبود توان عملیاتی تعداد زیادی تغییر کوچک استفاده می‌شود. بعد از اینکه نوشتن تمام شد، محتوای آن در memTable درج می‌شود.



شکل ۳- ۱۶: نمایش تبلت

وقتیکه یک عمل خواندن به یک سرور تبلت می‌رسد، بطور مشابه well-form بودن و مجاز بودن چک می‌شود. یک خواندن معتبر روی نمای ادغام شده رشته ای از SSTable ها و memTable اجرا می‌شود. چون SSTable و memTable ساختمان داده‌هایی هستند که بطور محلی (lexicographic) مرتب شده اند، نمای ادغام شده می‌تواند بطور کارامدی جواب بدهد.

عملیات خواندن و نوشتن ورودی می‌توانند ادامه داشته باشند تا زمانیکه تبلت‌های تقسیم و ادغام شوند.

به هم چسباندن Compaction :وقتی عملیات نوشتن اجرا می‌شوند، اندازه memTable افزایش می‌یابد. وقتیکه اندازه memTable به یک حد آستانه رسید، memTable قفل می‌شود، یک memTable جدید ایجاد می‌شود، و memTable قفل شده به یک SSTable تبدیل شده و در GFS نوشته می‌شود. این پردازه چسباندن جزئی[1] دو هدف دارد: نحوه استعمال حافظه سرور تبلت را کم[2] می‌کند، و میزان داده‌هایی که باید از commit log در طول بازیابی اگر این سرور بمیرد خوانده شود را کاهش می‌دهد.

هر چسباندن جزئی یک SSTable جدید ایجاد می‌کند. اگر این رفتار بدون چک ادامه داشته باشد، ممکن است عملیات خواندن نیازمند ادغام بروزرسانی‌ها از تعداد دلخواهی SSTable شوند. در عوض ما تعداد این فایل‌ها را با جرای متناوب merging compaction در پس زمینه محدود می‌کنیم. یک به هم چسباندن ادغام، محتووای چند SSTable و memTable را می‌خواند و در یک SSTable جدید می‌نویسد. SSTable هاو memtable ورودی بعد از اتمام compaction می‌توانند نادیده گرفته شوند.

یک چسباندن ادغام[3] که همه SSTable ها را در دقیقاً یک SSTable می‌نویسد به هم چسباندن کلی[4] نامیده می‌شود. SSTable هایی که توسط چسباندن‌های جزئی ایجاد می‌شوند می‌توانند شامل ورودی‌های حذف شده‌ای باشند که داده‌های حذف شده در SSTableهای مسن‌تر که هنوز زنده هستند را suppress می‌کند. چسباندن کلی، یک SSTable می‌سازد که شامل اطلاعات حذف (deletion) یا داده‌های حذف شده نیست. BigTable در طول همه تبلت‌ها می‌چرخد و معمولاً چسباندن کلی را به آنها اعمال می‌کند. این چسباندن کلی به BigTable اجازه می‌دهد تا منابعی که توسط داده‌های حذف شده استفاده شده‌اند را احیاء کند، و به آن اجازه می‌دهد تا اطمینان دهد که داده‌های حذف شده از سیستم در طول زمان ناپدید می‌شوند، که برای سرویس‌هایی که داده‌های حساس را ذخیره می‌کنند مهم است.

[1] minor compaction
[2] shrink
[3] merge compaction
[4] major compaction

پالایش Refinement:پیاده‌سازی توصیف شده در بخش قبلی نیازمند پالایش است تا به کارایی بالا، دسترس‌پذیری بالا، و قابلیت اطمینان بالای درخواست شده توسط کاربران برسد. این بخش پیاده‌سازی را با جزئیات بیشتر توصیف می‌کند تا این پالایش‌ها را برجسته کند.

گروه‌های محلی (Locality Groups): کلاینت‌ها می‌توانند چند خانواده ستون را با هم در یک گروه محلی، گروه بندی کنند. یک SSTable مجزا برای هر گروه محلی در هر تبلت ذخیره می‌شود. تفکیک خانواده‌های ستونی که معمولاً با هم در گروه‌های محلی مجزا دسترسی نمی‌شوند خواندن‌های کارامدتری ایجاد می‌کند. به عنوان مثال، متادیتای صفحه در Webtable (مثل زبان و chechsumها) می‌توانند در یک گروه محلی باشند و محتوای صفحه می‌توانند در یک گروه مختلف باشند: کاربردی که می‌خواهد متادیتا را بخواند نیازی ندارد تا همه محتوای صفحات را بخواند.

به علاوه، یکسری پارامتر تنظیم مفید می‌تواند برای هر گروه محلی تعیین شون. به عنوان مثال، می‌توان یک گروه محلی را طوری تعریف کرد که در حافظه RAM قرار گیرد. SSTableهای گروه‌های محلی مقیم در حافظه بصورت lazy در حافظه سرور تبلت بار می‌شوند. وقتی بارگذاری شدند، خانواده‌های ستونی که درچنین گروه‌های محلی قرار می‌گیرند می‌توانند بدون دسترسی به دیسک خوانده شوند. این ویژگی برای بخش‌های کوچکی از داده که به تناوب دسترسی می‌شوند مفید می‌باشد: ما آن را بطور داخلی برای مکان خانواده ستون در جدول متادیتا استفاده می‌کنیم.

فشرده سازی Compression :کلاینت‌ها می‌توانند کنترل کنند که چه موقع یا نه SSTable ها برای یک گروه محلی فشرده شوند و در صورت فشرده سازی از چه قالبی استفاده می‌شود. قالب فشرده سازی تعیین شده–توسط کاربر به هر بلوک SSTable اعمال می‌شود (که اندازه آن از طریق پارامتر گروه محلی قابل کنترل می‌باشد). هرچند که ما با فشرده سازی هر بلوک بطور مجزا مقداری فضا از دست می‌دهیم، اما مزیتی که بدست می‌آوریم این هست که بخش‌های کوچک SSTable می‌تواند بدون بازکردن[1] کل فایل خوانده شود.

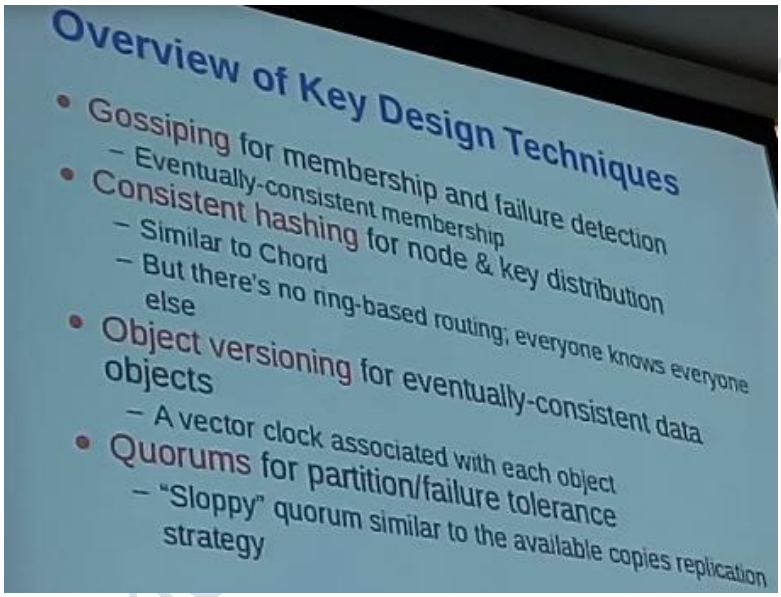
کش کردن برای بالا بردن کارایی خواندن: برای بهبود کارایی خواندن، سرورهای تبلت دو سطح کش را استفاده می‌کنند. Scan Cache یک کش سطح بالا می‌باشد که زوج‌های (کلید–مقدار) بازگشتی توسط رابط SSTable به کد سرور تبلت را کش می‌کند. Block Cache یک کش سطح پایین می‌باشد که بلوک‌های SSTable را که از GFS خوانده می‌شوند کش می‌کند. Scan Cache برای کاربردهایی که تمایل به خواندن متناوب داده‌های مشابه دارند خیلی مناسب است. Block cache برای کاربردهایی مناسب است که تمایل به خواندن داده‌هایی دارند که نزدیک داده‌هایی هستند که اخیراً خوانده‌اند. (مثلاً، خواندن‌های ترتیبی، یا خواندهای تصادفی ستون‌های مختلف در یک گروه محلی با یک سطر مهم)

Bloom Filter: همانطور که گفتیم، یک عمل خواندن باید همه SSTableهایی که حالت یک تبلت را می‌سازند را بخواند. اگر این SSTable ها در حافظه RAM نباشند، باید دسترسی‌های زیادی به دیسک داشته باشیم. ما می‌توانیم تعداد دسترسی‌ها را کاهش دهیم با اجازه به کلاینت‌ها که Bloom Filter هایی برای SSTableهای گروه‌های محلی خاص تعیین کنند. یک Bloom Filter به ما اجازه می‌دهد بپرسیم آیا یک SSTable شامل داده‌ای برای یک زوج سطر/ستون هست یا خیر. برای کاربردهای خاص، میزان کمی از حافظه سرور تبلت برای ذخیره Bloom Filter استفاده می‌شود که به شدت میزان دسترسی به دیسک را در عملیات خواندن کاهش می‌دهد. همچنین در هنگام جستجو برای سطر و ستون‌هایی که وجود ندارند نیاز به دسترسی به دیسک نداریم.

[1] Decompress

Dynamo -۳

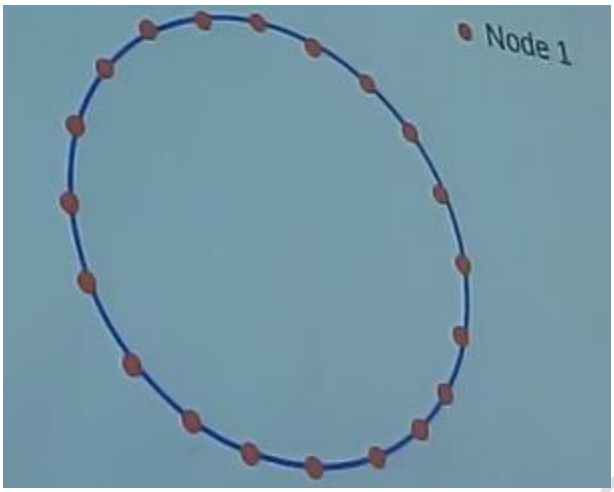
Dynamo توسط آمازون توسعه یافت. هدف نگه داری مقیاس‌پذیر داده‌ها و سرعت بالا بود. داینامو نیز یک مخزن ذخیره سازی key-value توزیع شده است و برای کاربری‌های مختلف آمازون استفاده می‌شود. دسترس پذیری بالا در داینامو برای آمازون خیلی مهم بود. موقع نوشتن داده‌ها را با سرعت می‌نویسیم و نگران consistency نیستیم ولی در موقع خواندن سازگاری را چک می‌کنیم. در عمل موقع خواندن داده‌ها سازگار می‌شوند (eventual consistency). هدف نوشتن بصورت peer-to-peer می‌باشد. هر داده‌ای که داریم را هش می‌کنیم و براساس مقدار هش آن تصمیم می‌گیریم که در کدام سیستم نوشته شود. همه سیستم‌ها روی روش هش توافق دارند. برای هر سیستم یک دامنه هش در نظر می‌گیرند و هر دیتایی که هشش در محدوده آن سیستم بود در آن ذخیره می‌شود. برای نسخه‌های مختلف داده شماره آخرین تغییر را نگه می‌دارد. برای تحمل پذیری خطا از Quorum استفاده می‌شود.(شکل ۳- ۱۶)



شکل ۳-۱۶ مرور روش‌های طراحی کلید داینامو

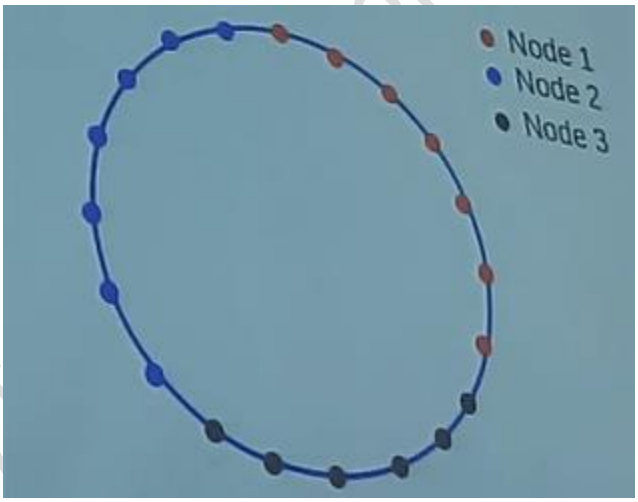
همانطور که در شکل ۳-۱۷ نشان داده شده است در داینامو گره‌ها بصورت حلقوی قرار می‌گیرند. البته منظور هم‌بندی[1] حلقوی نیست بلکه از لحاظ منطقی چنین ساختاری دارند و هر گره یک شماره هش دارد وقتیکه یک داده هش شد به گرهی ارسال می‌شود که هش داده در محدوده هش گره قرار گیرد و اولین گره موظف هست داده را در دو گره متوالی بعدی نیز کپی کند (Replica). (معمولاً گره بعدی در حلقه در رک دیگری قرار دارد).

[1] Topology



شکل ۳– ۱۷ ساختار گره‌های داینامو

اضافه و حذف کردن گره: برای اینکه ساختار حلقه بهم نریزد تعدادی گره مجازی[1] ایجاد می‌شود و هر گره جدید در یکی از این مجموعه گره مجازی قرار می‌گیرد. مثلا در شکل ۳– ۱۸ سه مجموعه گره مجازی داریم که گره ها در آن قرار می‌گیرند در اینصورت گره بعدی در مجموعه گره بعدی خواهد بود.

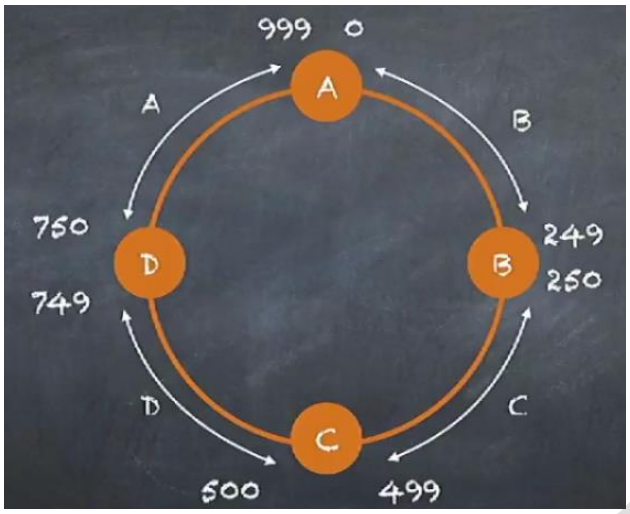


شکل ۳–۱۸: ساختار گره‌های مجازی در داینامو. در این شکل سه نوع گره مجازی داریم که با رنگ های متفاوت نشان داده شده‌اند. اگر یک کامپیوتر جدیدی اضافه شد یکی از گره‌های مجموعه گره مجازی را به آن اختصاص می‌دهیم.

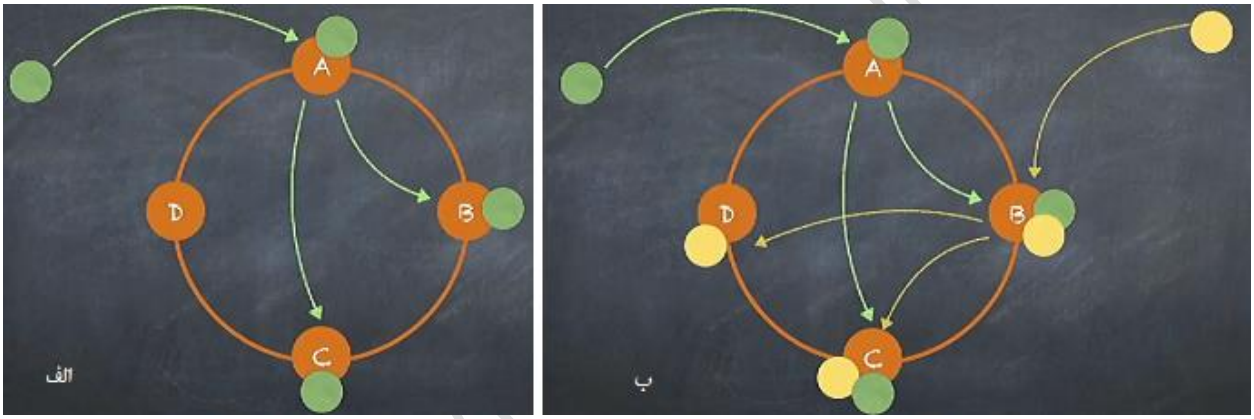
در Hdfs فایل‌ها بصورت توزیع شده قرار می‌گیرند ولی در داینامو key-value ها بصورت توزیع شده قرار گیرند.

اجازه دهید ساختار داینامو را با جزئیات بیشتر توضیح دهیم. همانطور که در شکل ۳– ۱۹ نشان داده شده است به گره‌ها یک دامنه هش اختصاص داده شده است و هر داده ای که در آن دامنه هش قرار گیرد در آن گره ذخیره خواهد شد. مهم نیست ما با چه گرهی حرف بزنیم و داده ها را به آن بفرستیم آن گره خودش داده‌ها را به گره با هش مرتبط می‌فرستد.

[1] Virtual node

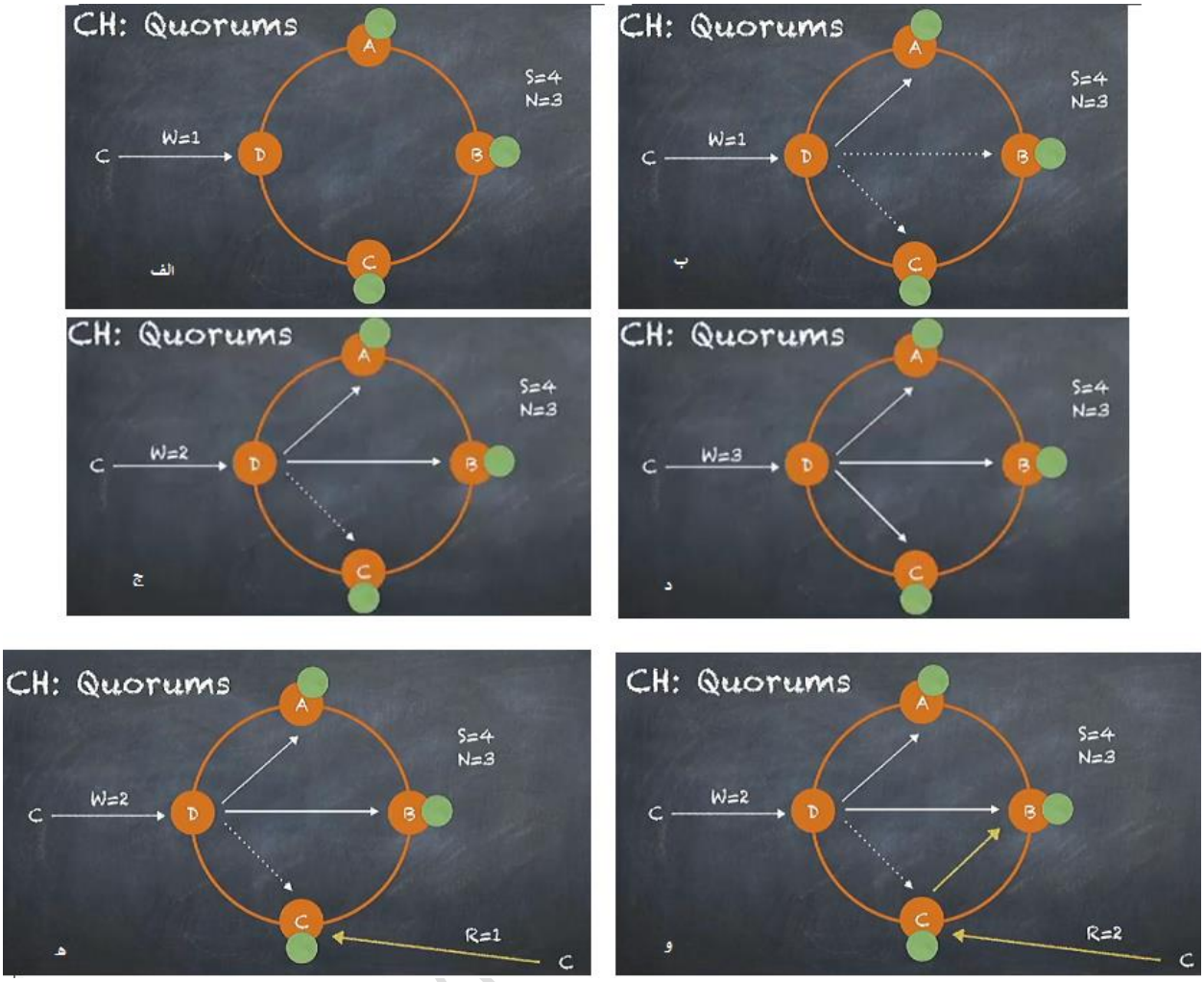


شکل ۳- ۱۹: همانطور که می‌بنید به هر گره یک دامنه هش اختصاص داده شده است مثلا گره B دامنه ۰ تا ۲۴۹ سیستم C دامنه ۲۵۰ تا ۴۹۹، سیستم D دامنه ۵۰۰ تا ۷۴۹ و A دامنه ۷۵۰ تا ۹۹۹ دارد.



شکل ۳- ۲۰: روش ذخیره داده‌ها در داینامو

در شکل ۳-۲۰ روش ذخیره سازی در داینامو نشان داده شده است. در شکل ۳-۲۰-الف یک داده که با رنگ سبز وارد شده است در گره A براساس هش هر دو و تطابق آنها ذخیره می‌شود و گره A به عنوان اولین گره آن را در دو گره بعد در حلقه یعنی گره‌های B و C ذخیره می‌کند. به همین طریق در شکل ۳-۲۰-ب داده جدیدی که با رنگ زرد نشان داده شده است قرار است بر اساس هش در گره B نوشته شود. گره B نیز مسئول کپی آن در گره‌های C,D می‌باشد. در داینامو می‌توان تعداد Replica ها را مشخص کرد که البته بطور پیش‌فرض هر داده سه بار و در سه گره متوالی ترجیحاً در رک‌های مختلف نوشته می‌شود. این باعث بالارفتن دسترس پذیری می‌شود و در صورت خرابی یک رک داده‌ها در رک دیگری قرار دارند.

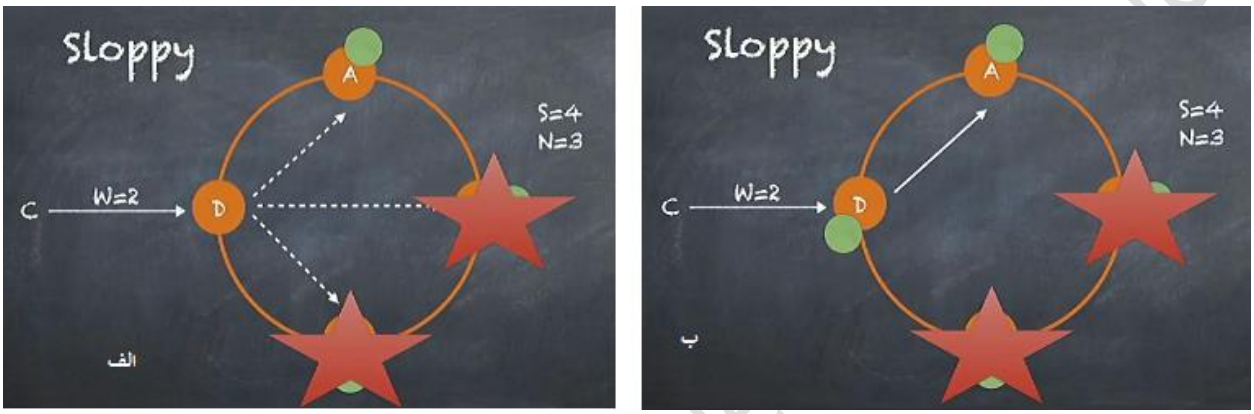


شکل۳- ۲۱: خواندن و نوشتن در داینامو

در شکل ۳-۲۱ روش خواندن و نوشتن در داینامو توضیح داده شده است. در شکل S=4 عدد سرور داریم و تعداد کپی‌ها N=3 می‌باشد. می‌خواهیم موقع نوشتن یک پارامتر به نام w را کنترل کنیم. W تعیین می‌کند برای نوشتن منتظر w تا سرور هستیم که از ۳ عدد کپی پاسخ (acknowledge) بدهند. هر چه w کمتر باشد یعنی consistency کمتر است. در شکل ۲۱-الف w=1 بنابراین وقتی یکی از کپی ها پاسخ داد منتظر پاسخ دو کپی دیگر نمی‌مانیم. در شکل ۳- ۲۱-ج w=2 یعنی consistency را با پاسخ دو سرور replica تنظیم می‌کنیم و وقتی w=3 به حداکثر consistency می‌رسیم. در دو تای قبل حداکثر سازگاری وجود ندارد و eventual consistency بدست می‌آید. بطور مشابه هنگام خواندن یک پارامتر به نام R داریم که تعیین می‌کند هنگام خواندن چند تا از سرورهای replica جواب بدهند. در شکل ۳-۲۱-ه w=2 و R=1 می‌باشد. همانطور که مشاهده می‌کنید داده در دوتا از گره‌های A,B ذخیره شده است ولی هنوز با C سازگار نشده است. بنابراین اگر در خواندن ابتدا به C رجوع کنیم ممکن است داده سازگار با دو کپی دیگر نداشته باشیم. بهترین انتخاب این است که W+R=N+1 باشد تا طبق اصل لانه کبوتر حتما یکی از داده‌ها در دو خانه مشترک قرار داشته باشند که در شکل ۳- ۲۱-و نشان داده شده است. دو پارامتر W,R قابل کنترل است و برای کاربردهایی که نیازمند consistency بالا هستند w را بالاتر در نظر می‌گیریم تا اطمینان از نوشتن در چند کپی بدهیم. اما در عوض دسترس پذیری کمتر می‌شود. قاعدتاً سرعت وقتی w زیاد است کمتر خواهد شد. به این قابلیت یعنی کنترل میزان تایید نوشتن و خواندن Quorum (حدنصاب) می‌گویند.

Sloppy Quorum

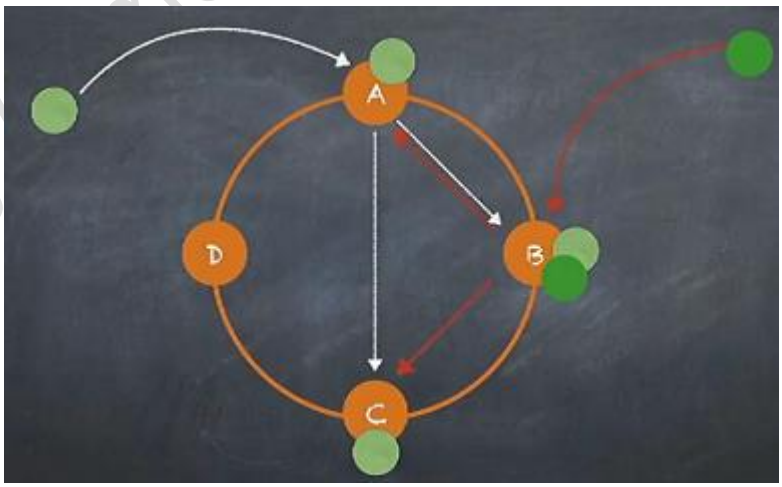
گاهی اوقات هنگام نوشتن یا خواندن ممکن است همه گره‌های متوالی در دسترس نباشند مثلا busy باشند یا قطع شده باشند. در این حالت گره اصلی داده را نگه می‌دارد وقتی که در دسترس شدند آنها را می‌فرستد. (lazy) . همانطور که در شکل ۳- ۲۲-الف مشاهده می‌شود داده قرار هست در گره‌های A و B و C نوشته شود و پارامتر w=2 می‌باشد. در این حالت فقط A در دسترس است داده در A نوشته می‌شود (شکل۳-۲۲-ب). اما چون B و C در دسترس نیستند آن را به گره D فرستاده و ذخیره می‌کند در این روش لزوماً داده‌ها ممکن است به رک های مختلف نروند. وقتی گره های Cو D مجدداً در دسترس قرار گیرند داده‌ها به آنها ارسال می‌شود.



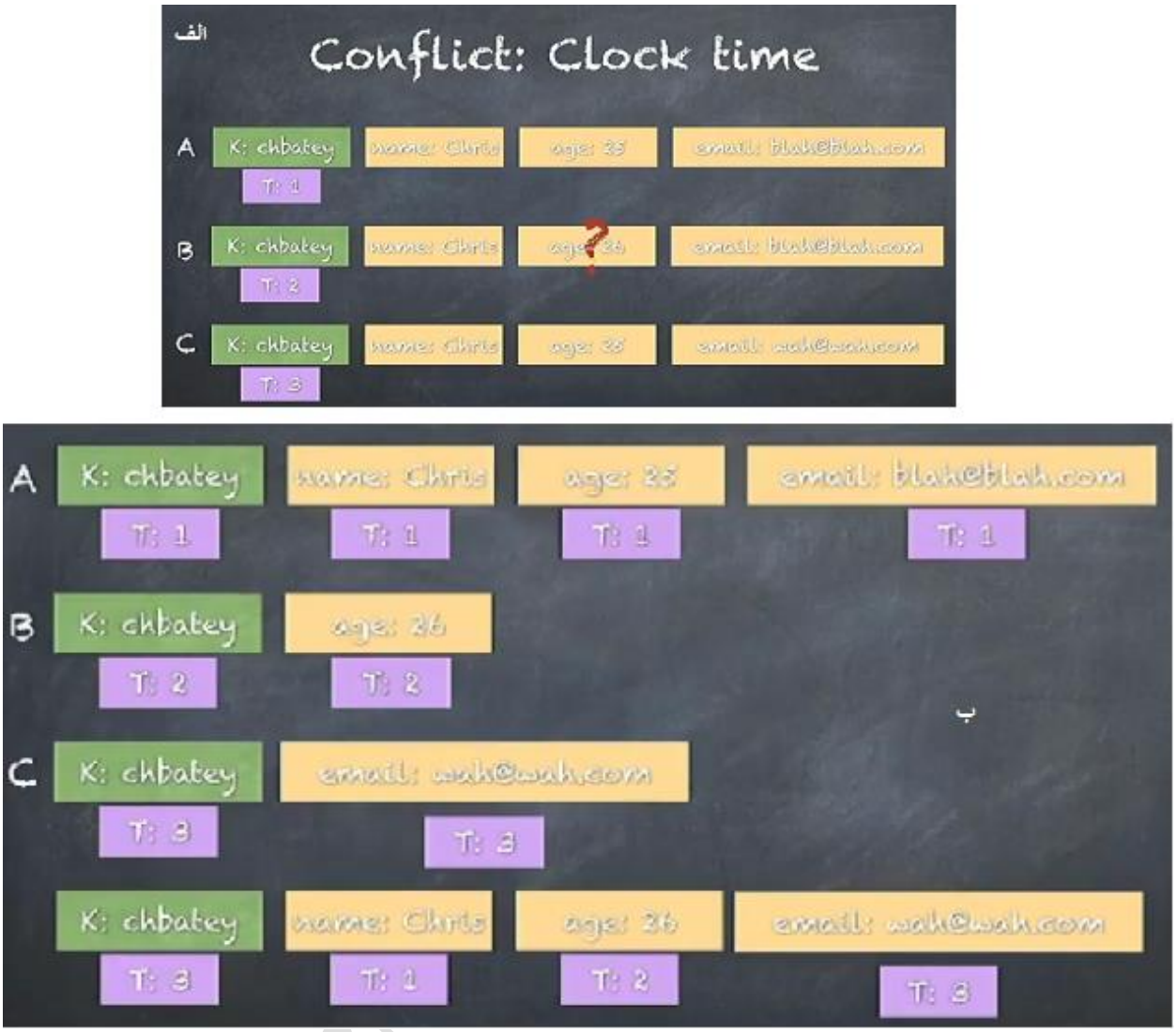
شکل ۳- ۲۲: sloppy Quorums

رفع و رجوع تصادم

اگر یک داده توسط یک کلاینت بروز شود و دیگری بخواهد یک کپی آن را از گره دیگری بخواند چه پیش خواهد آمد؟ conflict چگونه حل می‌شود؟ در پایگاه داده‌های سنتی حتماً باید consistency برقرار باشد که این عمل با قفل کردن مکان دسترسی مشترک ایجاد می‌شود. در dynamo همه اجازه نوشتن دارند ولی موقع خواندن conflict را رفع و رجوع می‌کنیم (شکل ۳-۳۲). در داینامو همیشه آخرین نسخه‌ای که روی پایگاه داده نوشته شده است معتبر است.



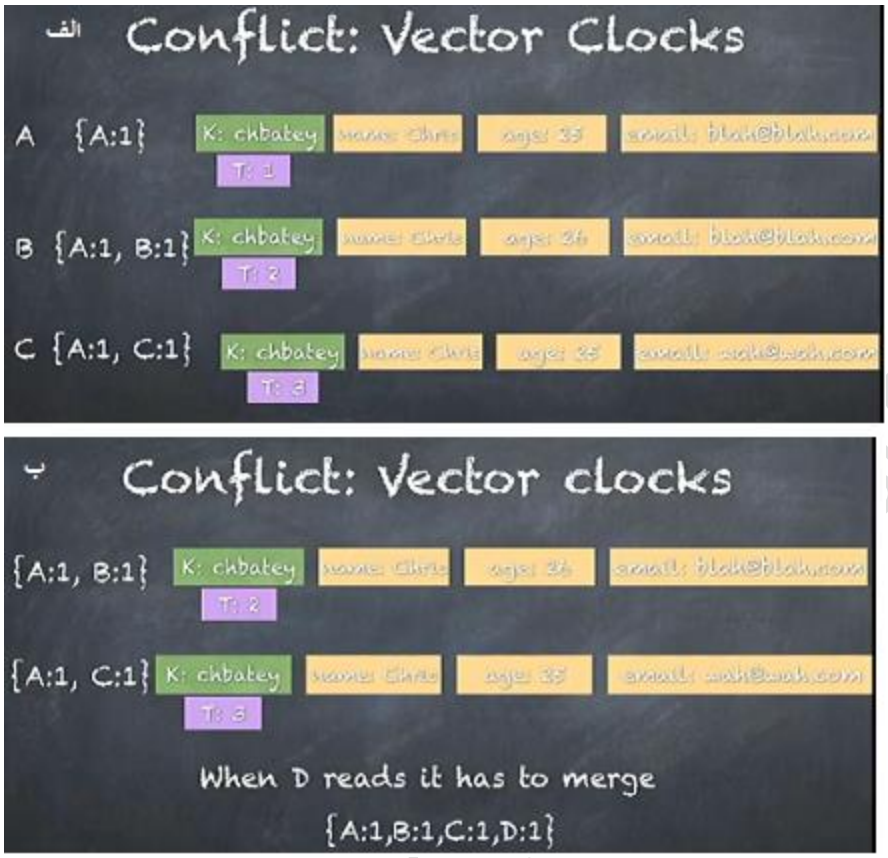
شکل ۳-۲۳-حل و فصل تصادم



شکل ۳-۲۴- حل و فصل تصادم با Column Familly

در شکل ۳-۲۴ حالت دیگری از حل و فصل تصادم نشان داده شده است. سه کاربر A و B و C را در نظر بگیرید که روی یک سطر داده کار می کنند همانطور که در شکل ۳-۲۴-الف مشاهده می‌کنید کاربر A در T:1 سطر را تغییر داده است. کاربر B در T:2 سن این سطر را عوض کرده است و کاربر C در T:3 ایمیل کاربر را تغییر داده است. اما چون داینامو همیشه آخرین تغییر را معتبر می‌داند داده C را برمی‌گرداند و تغییرات B مشاهده نخواهد شد. راه حل همانطور که در شکل ۳-۲۴ج نشان داده شده است استفاده از Column family است که تغییرات براساس ستون باشد در اینصورت مساله رفع و رجوع می‌شود. که البته این هنوز هم مشکل خواهد داشت اگر دو کاربر به یک column family دسترسی داشته باشند. که می‌توان برای هر column family قفل گذاشت که این باعث پایین آمدن availability (دسترس پذیری) می‌شود.

نکته‌ای که ممکن است در اینجا بوجود بیاید عدم sync بودن کلاک‌ها در سرورهای مختلف می‌باشد. روش دیگر حل و فصل کردن تصادم استفاده از clock vector است که هم زمان تغییر و هم نویسنده تغییر را نگه می‌دارد. که در شکل ۳-۲۵ نشان داده شده است:



شکل ۳-۲۵- حل و فصل conflict با بردار ساعت

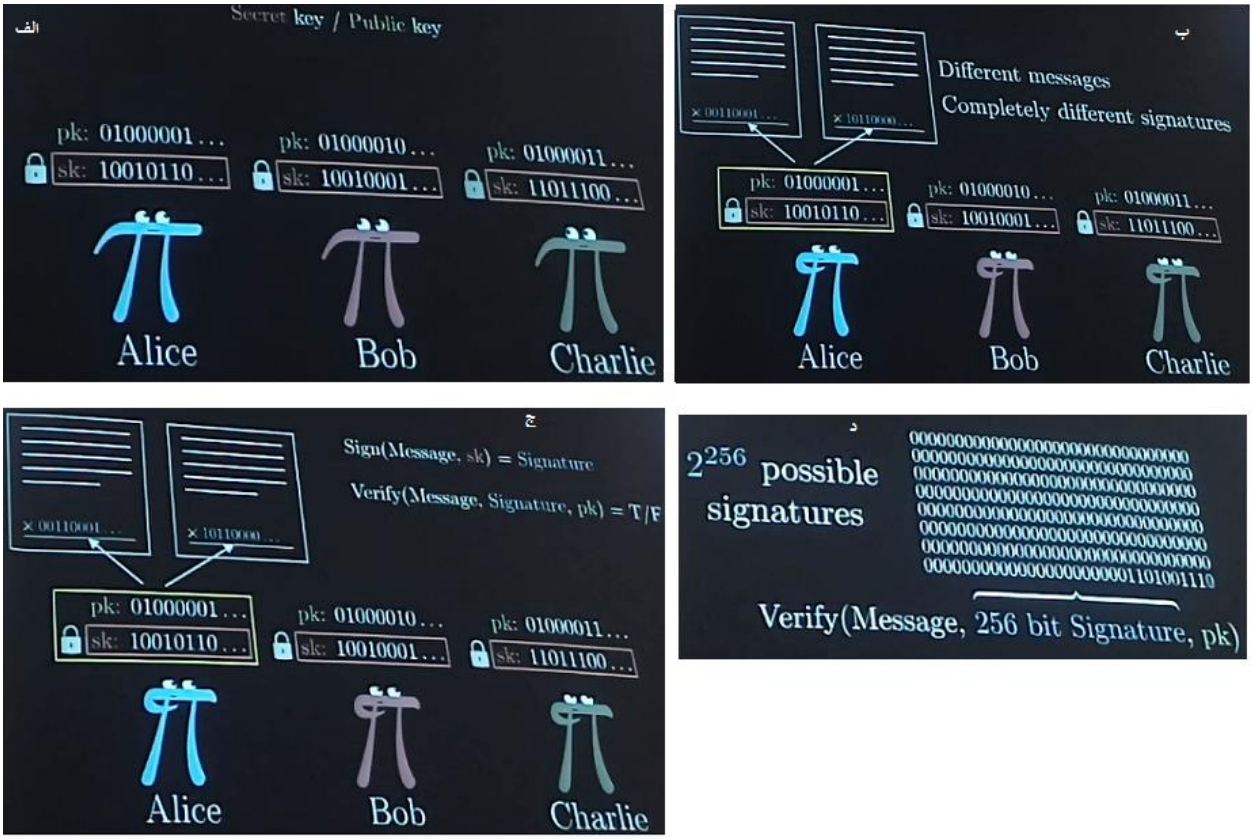
همانطور که در شکل ۳-۲۵-ف مشاهده می شود به ازای هر تغییر هم نویسنده تغییر و هم زمان تغییر را بصورت بردار داریم. در شکل الف سیستم A در زمان ۱ تغییر داده است و بصورت بردار {A:1} نشان داده شده است. سیستم B و C تغییر A را دیده اند و تغییرات خود را نیز داده اند. اما سیستم C تغییر B را ندیده است. سیستم D هر دو بردار B,C را می‌خواند و آنها را باهم ادغام می‌کند و بردار تغییرات ادغام شده را می‌نویسد.

مقایسه BigTable با Dynamo: در Bigtable به خاطر حفظ محلیت بین تبلت ها اسکن کردن در گوگل سریعتر است اما داینامو دسترس پذیری داده دارد. در آمازون فقط 0.1 درصد حالات conflict دارد. داینامو راجع به توپولوژی دیتاسنتر های هم مطالبی در اختیار ما قرار می‌دهد.

Bit Coin and Block Chain-۴

بصورت سنتی یکسری currency وجود دارد که از قدیم به عنوان واحد پول مطرح بودند و در تراکنش‌های مختلف توسط انسان‌ها استفاده می‌شوند. در آن زمان سیستم‌های مالی نامتمرکز بود و با ایجاد بانک ها و بانک مرکزی به سیستم های متمرکز تبدیل شد. اسکناس ها ذاتاً ارزشی ندارند ولی بانک مرکزی آنها را تایید می‌کند که خود بانک مرکزی SPOF می‌باشد. در سال ۲۰۰۸ در آمریکا یک مشکل مالی مطرح شد که تحت عنوان Bitcoin: A Peer-to-Peer Electronic Cash System ارائه شد. BitCoin برای این مطرح شد که سیستم مالی متمرکز نباشد. خود Bitcoin یک واحد پول و ارزش هست. Blockchain راه حلی ارائه می‌دهد که تک تک افراد می‌توانند قصد خرابکاری داشته باشند ولی کل سیستم درست کار می‌کند. تعداد زیادی (چند هزار) واحد پول مجازی هست که در مقابل آنها نه پولی رد و بدل می‌شود نه طلا و نه چیز دیگری فقط اعتبار جابجا می‌شود. (مثل Bitcoin,Ethereum,Ripple, Litecoin, Ethereum Classic).

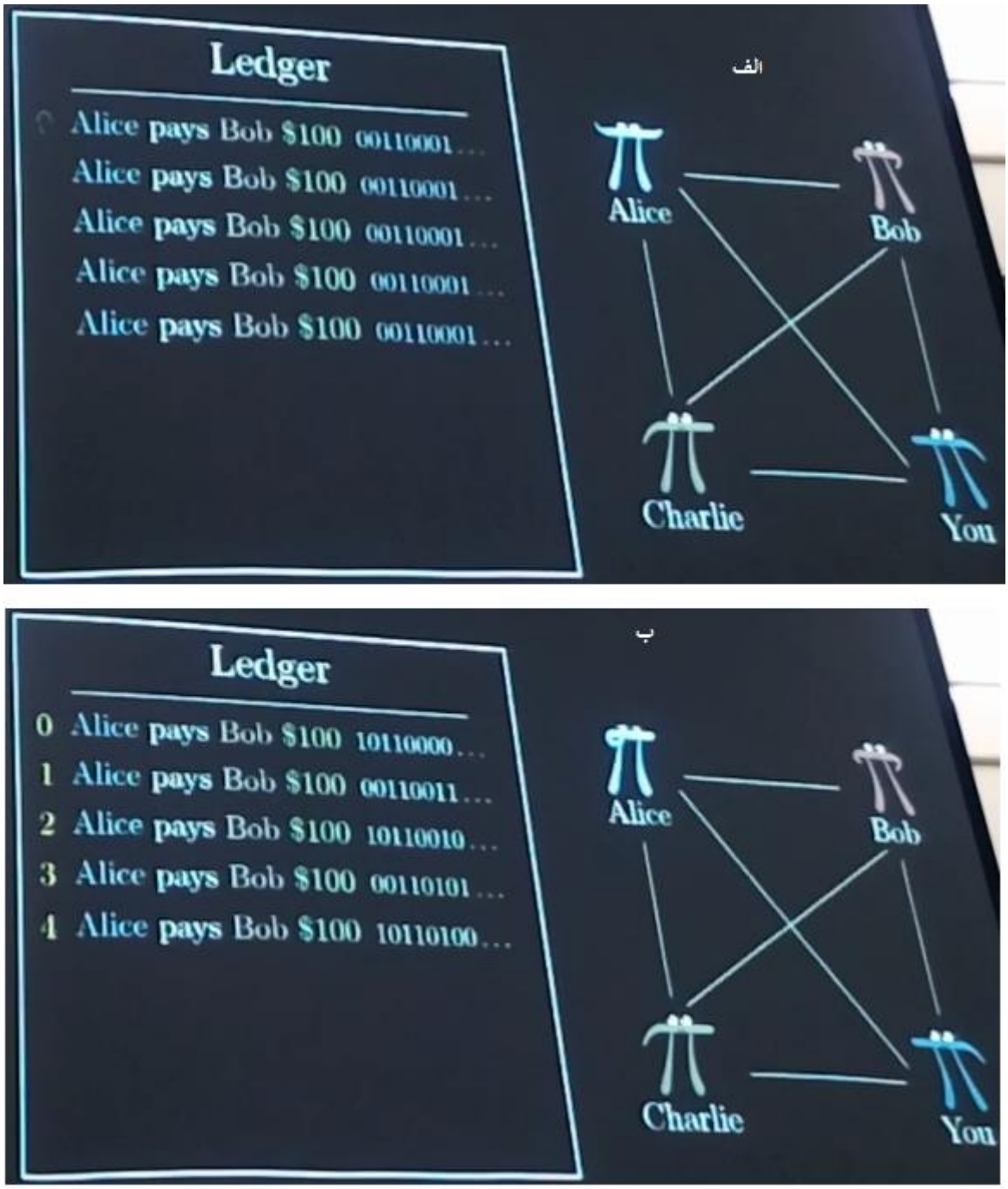
Bitcoin هم مثل سایر موارد بانکی یکسری پروتکل دارد که نیازی نیست استفاده کنندگان از آن بدانند چه عملیاتی در آن اتفاق می‌افتد. در مفهوم بیت کوین یکسری دفتر کل ledger وجود دارد که وضعیت تراکنش ها را نشان می‌دهد برای تعیین صحت این تراکنش ها باید از روشهایی مثل امضای دیجیتال استفاده کرد. بطوریکه فرد پرداخت کننده بتواند صحت آن را امضا کند. . برای ایجاد امضای دیجیتال از کلید عمومی و کلید خصوصی (کلید سّری) می‌توان استفاده کرد که روند ایجاد آن در شکل ۳-۲۶ نشان داده شده است.



شکل ۳-۲۶: امضای دیجیتال

همانطور که در شکل ۳-۲۶-الف مشاهده می کنید برای ایجاد امضای دیجیتال از کلید عمومی pk و کلید سّری Sk استفاده می‌شود. کلید سری فقط توسط شخص استفاده کننده استفاده می‌شود و بقیه آن را نمی بینند و هویت آن شخص است و باید محافظت شود. کلیدهای خصوصی منحصر بفرد هستند. در شکل ۳-۲۶-ب مشاهده می کنید که برای ایجاد امضای دیجیتال باید پیام های مختلف امضای مختلف داشته باشند در غیر اینصورت امضای هر فرد را می‌توان زیر هر متنی کپی کرد. برای جلوگیری از این عمل در شکل ۳-۲۶-ج نشان داده شده است که امضا از متن پیام و کلید سری تولید می‌شوند. Sign(Message, sk)=signature. بنابراین حتی اگر یک بیت دو پیام مختلف باشد امضاهای مختلفی ایجاد خواهد شد. اما پردازه تصدیق می‌تواند توسط هر فردی با کلید عمومی انجام شود. Verify(Message,Signature,pk)=T/F هر کس می‌تواند تایید کند با کلید عمومی که آیا این امضا صحیح هست یا خیر؟ اما کسی نمی تواند از روی امضا کلید خصوصی را بدست آورد. امضا ۲۵۶ بیت است و بنابراین 2^{256} امضای مختلف قابل تولید است که هک کردن آن کار سختی است.

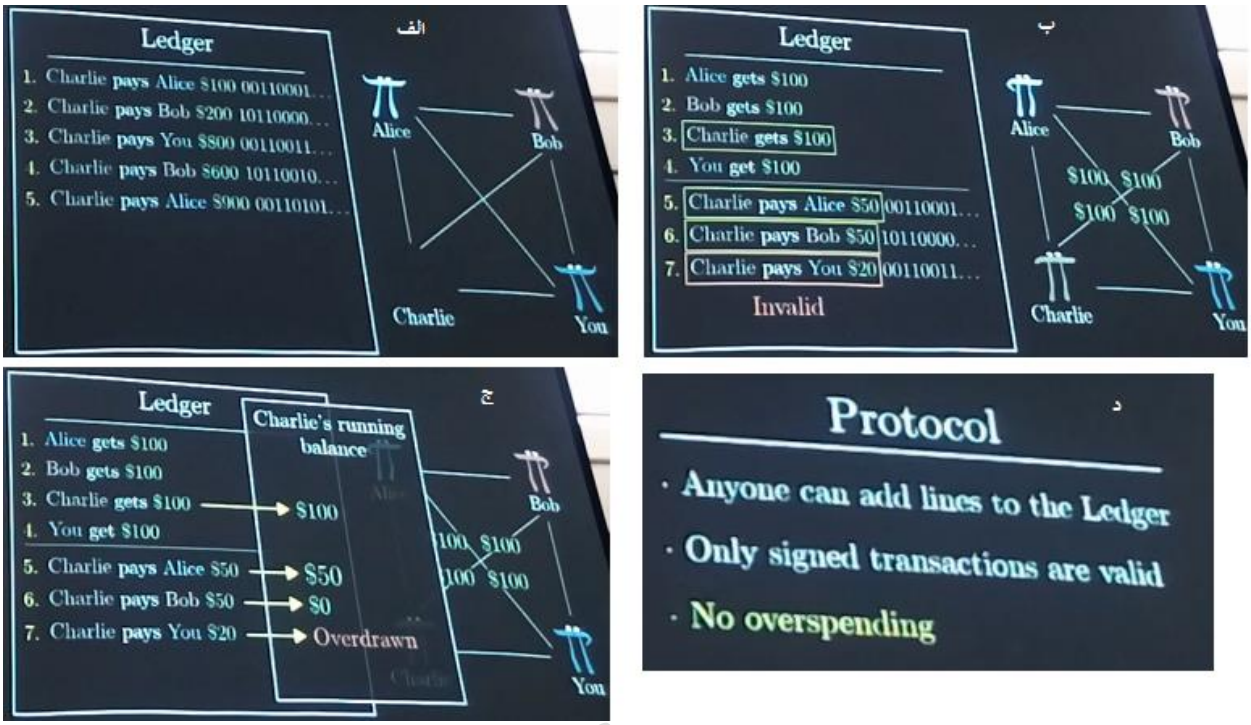
نکته ای که در این امضا ممکن است پیش بیاید این هست که کل پیام و امضا با هم کپی شوند و پیام تکرار شود. برای جلوگیری از این کار دو روش می‌توان استفاده کرد یکی مهر زمانی یا شماره پیام که در داخل پیام درج می‌شود و در ایجاد امضا تاثیر گذار است و دیگری قرار دادن پیام ها در یک زنجیره متوالی می‌باشد. در شکل ۳-۲۷ این مورد توضیح داده شده است.



شکل ۳-۲۷: جلوگیری از کپی کردن امضا با مهر زمانی.

همانطور که در شکل ۳-۲۷-الف مشاهده می‌شود یک پیام چندین بار کپی شده است. برای جلوگیری از این کار در هر پیام یک شماره پیام اضافه شده است که در روند ایجاد امضا تاثیرگذار است و برای هر پیام امضای متفاوتی ایجاد شده است این در شکل ۳-۲۷-ب نشان داده شده است.

بنابراین قرارداد به این صورت تنظیم می‌شود: ۱- هر کسی می‌تواند به ledger یک خط اضافه کند. ۲- در آخر هر ماه صورت حساب ها تسویه می‌شوند. ۳- فقط پیام های دارای امضا معتبر هستند.



شکل ۳-۲۸- روش جلوگیری از بخشش از جیب خالی.

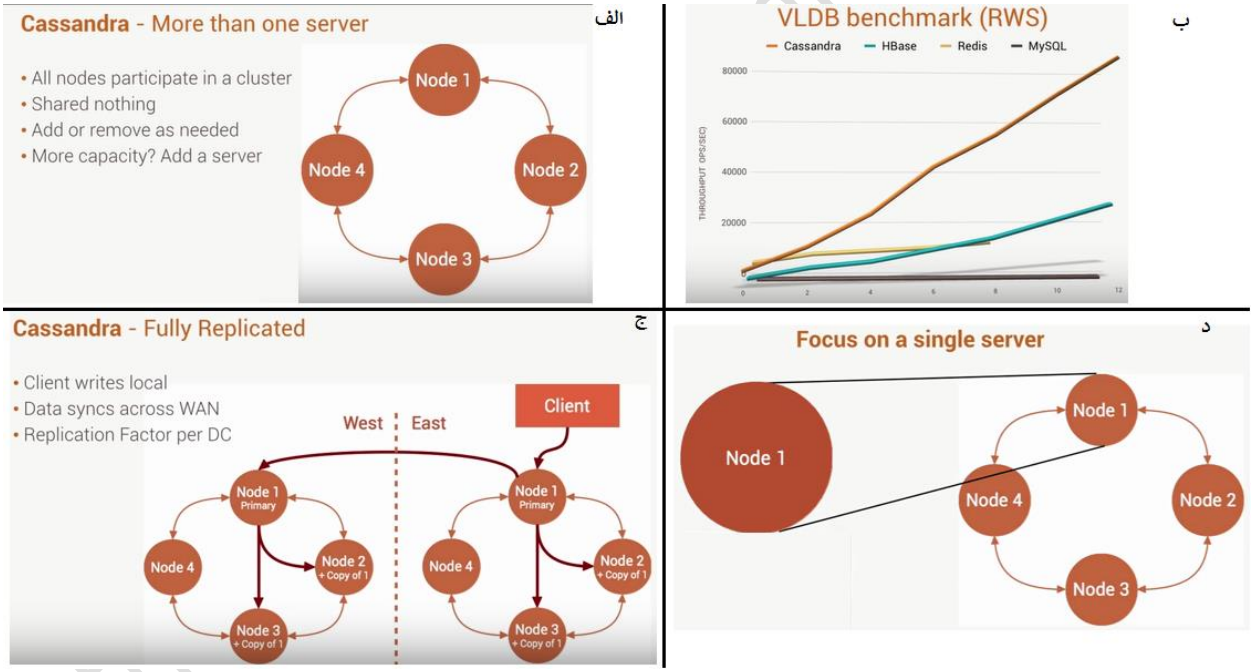
ایراد دیگر وارده بر این روش تاکنون این هست که فردی بخواهد بدون سپرده و موجودی پول خرج کند. مثلا در شکل ۳-۲۸-الف Charlie دارد بطور گسترده پول خرج می‌کند. آیا پول دارد؟ برای جلوگیری از این روش باید تاریخچه دفتر کل را بررسی کرد در صورتی که دخل و خرج با هم همخوانی نداشته باشد و موجودی فرد منفی شود تراکنش نامعتبر خواهد بود. در شکل ۳-۲۸-ب می‌بینید Charlie ۱۰۰ دلار گرفته بنابراین دو تراکنش اول او درست هست ولی تراکنش سوم نامعتبر است چون حساب او منفی می‌شود. در شکل ۳-۲۸-ج روش بررسی تاریخچه Charlie نشان داده شده است. بنابراین قرارداد بصورت شکل ۳-۲۸-د تغییر می‌یابد. ولخرجی با جیب خالی نداریم!!!

روش دیگر برای جلوگیری از کپی ایجاد یک زنجیر است که بلوک ها را در آن قرار می دهند و بلوک های کپی نمی توانند در زنجیره درست قرار گیرند این روشی است که Block chain استفاده می‌کند. در بلاک چین زنجیره‌ای از ledger ها بصورت متوالی قرار می‌گیرند که در انتهای آنها یک کد قرار دارد که هش آن باعث ایجاد تعدادی صفر در ابتدای هش (مثلا ۶۰ بیت صفر) می‌شود. بعد از ایجاد این کد ledger بسته شده و در انتهای زنجیر قرار می‌گیرد که به این عمل proof of work می‌گویند.

Cassandra-۴

Cassandra یک سیستم ذخیره سازی توزیع شده برای مدیریت مقدار بسیار زیاد داده‌های ساختاریافته توزیع شده در بین تعداد زیادی سرور می‌باشد، در حالیکه قابل دسترس بودن سرویس آن بالا است و هیچ نقطه شکست SPOFی ندارد. هدف کاساندرا اجرا روی یک زیرساخت با صدها گره می‌باشد (که احتمالاً در دیتاسنترهایی مختلف واقع شده‌اند.). در این مقیاس، اجزاء کوچک و بزرگ بطور متناوب دچار خطا می‌شوند. روش مدیریت حالت دائمی در مواجهه با این شکست‌ها، راه‌اندازی مطمئن و مقیاس‌پذیر سیستم‌های نرم‌افزاری روی این سرویس می‌باشد. کاساندرا مدل داده‌ی رابطه‌ای کامل را پشتیبانی نمی‌کند در عوض، برای کلاینت‌ها یک مدل داده ساده را تهیه می‌کند که کنترل پویای ساختار و قالب

داده‌ها را پشتیبانی می‌کند. کاساندرا طوری طراحی شده است که روی سخت افزارهای ارزان اجرا شود و بازدهی بالایی برای نوشتن داشته باشد ولی بازدهی خواندن‌ها را قربانی آن نکند. کاساندرا توسط فیس بوک توسعه داده شد. فیس بوک بزرگترین پلت فرم شبکه اجتماعی را اجرا می‌کند که صدها میلیون کاربر را در زمان‌های پیک با استفاده از هزاران سروری که در دیتاسنترهای زیادی در سراسر دنیا واقع شده‌اند سرویس می‌دهد. نیازهای عملیاتی سختگیرانه‌ای در پلتفرم فیس بوک براساس کارایی، قابلیت اطمینان و بازدهی، وجود دارند و برای پشتیبانی رشد مداوم پلتفرم باید مقیاس‌پذیری بالایی داشته باشد. در شکل ۳-۲۹ ساختار کاساندارا نشان داده شده است. در شکل ۳-۲۹-الف نشان داده شده است که چندین سرور در کاساندرا استفاده می‌شوند و در یک خوشه قرار دارند و در صورت لزوم سرورها حذف یا اضافه می‌شوند و برای ظرفیت بیشتر نیز سرور اضافه می‌شود که خود نشان دهنده مقیاس‌پذیری کاساندرا است. در شکل ۳-۲۹-ب با برنامه محک VLDB توان عملیاتی کاساندرا در مقایسه با HBASE,Radis, MySQL سنجیده شده است مشاهده می‌شود سرعت خواندن و نوشتن در کاساندرا خیلی بالا هست که این مقیاس پذیری خطی آن را نشان می‌دهد یعنی با افزایش سرورها کارایی بطور خطی افزایش می‌یابد. کاساندرا از معماری داینامو استفاده می‌کند و یکسری ویژگی‌ها به آن اضافه می‌کند. همانطور که در شکل ۳-۲۹-ج مشاهده می‌کنید کپی‌های داده‌ها براساس فاکتور دیتاسنتر تکرار می‌شوند و نکته مهمی که دارد این می‌باشد که رپلیکاها ممکن است در دیتاسنترهای مختلف باشند و بصورت آسنکرون کپی می‌شوند یعنی منتظر Acknowledge از سرورهایی که کپی‌ها را نگه خواهند داشت، نمی‌شود. در شکل ۳-۲۹-د نیز روی یک گره تمرکز کرده است و قصد داریم در ادامه روش نوشتن در یک گره خاص را توضیح دهیم.

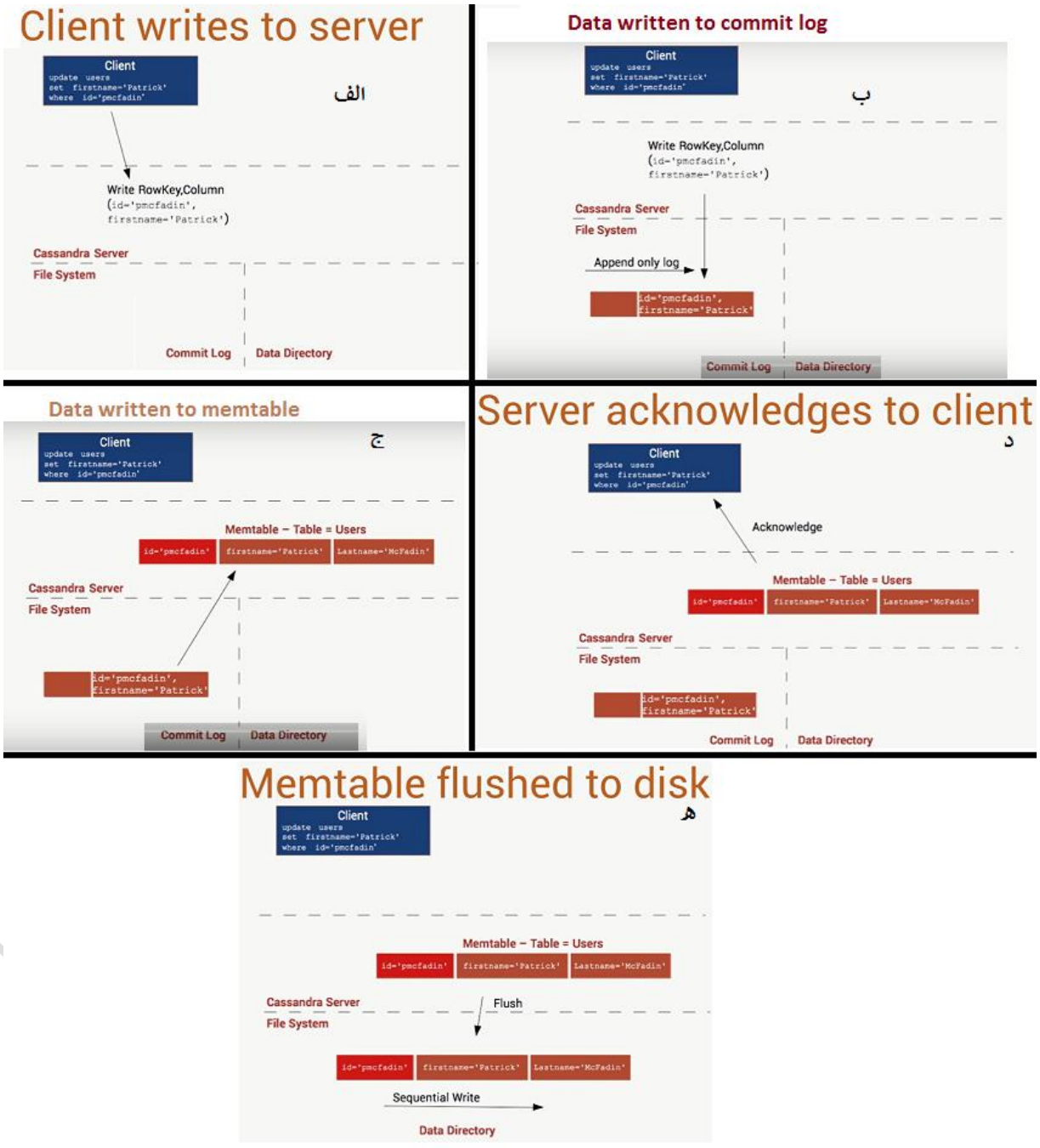


شکل ۳-۲۹: ساختار چند سرور کاساندرا و سنجش توان عملیاتی آن با چند سیستم داده‌ای دیگر

کاساندرا از نوع سیستم AP می‌باشد یعنی قابل دسترس بودن و تحمل پذیری پارتیشن دارد اما سازگاری را کامل برآورده نمی‌کند. در حقیقت Eventual Consistency دارد.

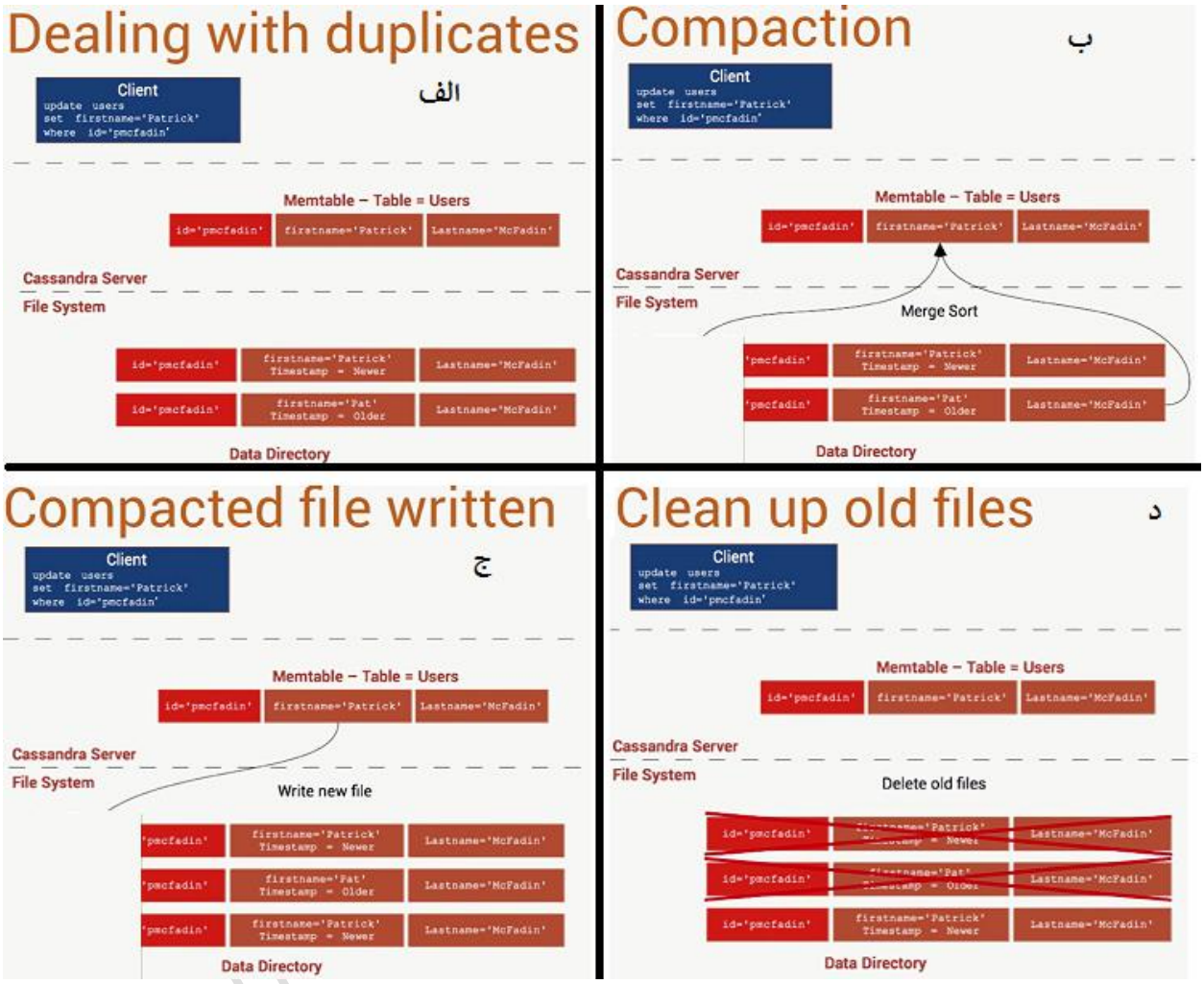
نوشتن یک کلاینت در سرور: همانطور که در شکل ۳-۳۰ الف مشاهده می‌کنیم کلاینت با دستور Update users set firstname='Patric' where id='pmcfadin' درخواست نوشتن به سرور می‌دهد. سیستم فایل شامل دو بخش Commit log و Data Directory می‌باشد که Commit Log نقش چرک نویس دارد. با توجه به شکل ۳-۳۰-ب درخواست کلاینت در commit log

نوشته می‌شود بصورت دائمی یعنی با قطع برق داده‌ها از بین نخواهند رفت ولی این مکان نهایی برای نوشتن داده‌ها نخواهد بود. هدف بالابردن سرعت نوشتن می‌باشد. سپس همانطور که در شکل ۳-۳۰-ج مشاهده می‌کنید داده‌ها از Commit log به memTable وارد می‌شوند یعنی به memory آورده می‌شوند و بعد از این مرحله به کلاینت Acknowledge داده می‌شود. در انتها بعد از اینکه حجم memTable به یک حد آستانه برسد داده‌ها بصورت ترتیبی در Data Directory (دیسک) نوشته می‌شوند که به این مرحله Flush می‌گوییم. کاساندرا از قابلیت مرتب سازی BigTable استفاده می‌کند و با یک مکانیزمی داده‌های memTable را بصورت مرتب نگه می‌دارد.



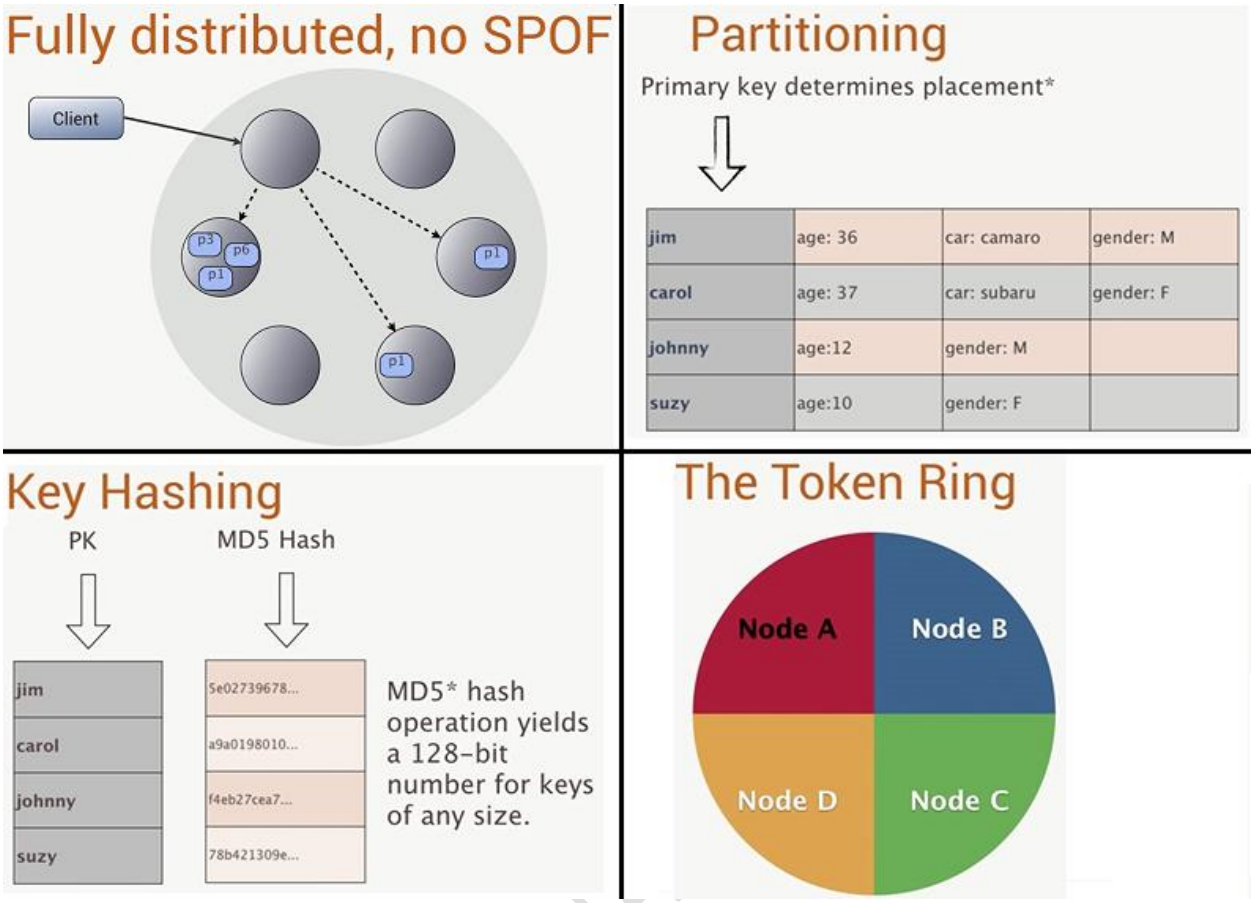
شکل۳-۳۰: در این شکل روند نوشتن در سرور را نشان داده ایم. همانطور که در شکل‌ها مشاهده می‌شود یک commit log داریم که نقش یک چرک نویس را بازی می‌کند و Data Directory برای ذخیره نهایی داده‌ها می‌باشد.

در شکل ۳–۳۱ روشی که کاساندرا با داده‌های تکراری کار می‌کند نشان داده شده است. همانطور که در شکل ۳–۳۱ –الف مشاهده می‌کنید دو داده مشابه در دیتا دایرکتوری وجود دارد که برای آنها مهر زمانی newer , Older تعریف شده است. در شکل ۳–۳۱–ب این دو داده به روش merge sort با هم در memTable مرتب و ادغام می‌شوند. که وقتی کپی ها تکرار می‌شود داده‌های قدیمیتر حذف می‌شوند که در شکل‌های ۳–۳۱–ج و د نشان داده شده است.



شکل ۳–۳۱ روش کار کردن کاساندرا با داده‌های تکراری در زمان‌های مختلف.

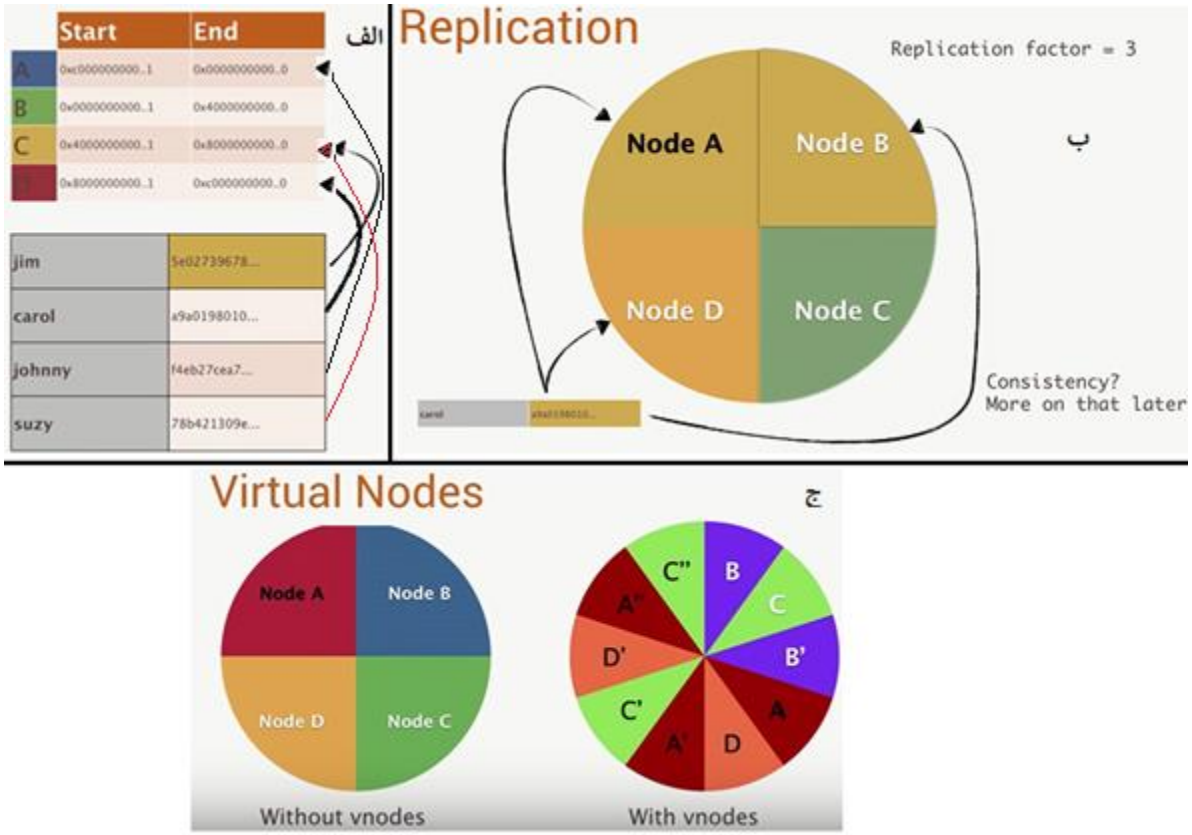
در کاساندرا SPOF نداریم زیرا سیستم کاملا توزیع شده است که این در شکل ۳–۳۲–الف نشان داده شده است. اما مکان ذخیره سازی مثل داینامو با هش تعیین می‌شود. داده‌ها براساس کلید هش می‌شوند و برای هر گره نیز یک مجموعه هش در نظر گرفته می‌شود و داده در گرهی قرار می‌گیرد که هش آن در رنج هش سرور قرار داشته باشد. همانطور که در شکل ۳–۳۲–د نشان داده شده است گره‌ها به روش توکن رینگ با هم در ارتباط می‌باشند.



شکل ۳-۳۲ : ساختار توزیع شده Cassandra و روش هش کردن داده‌ها در آن. ساختار گره‌ها بصورت توکن رینگ می‌باشد.

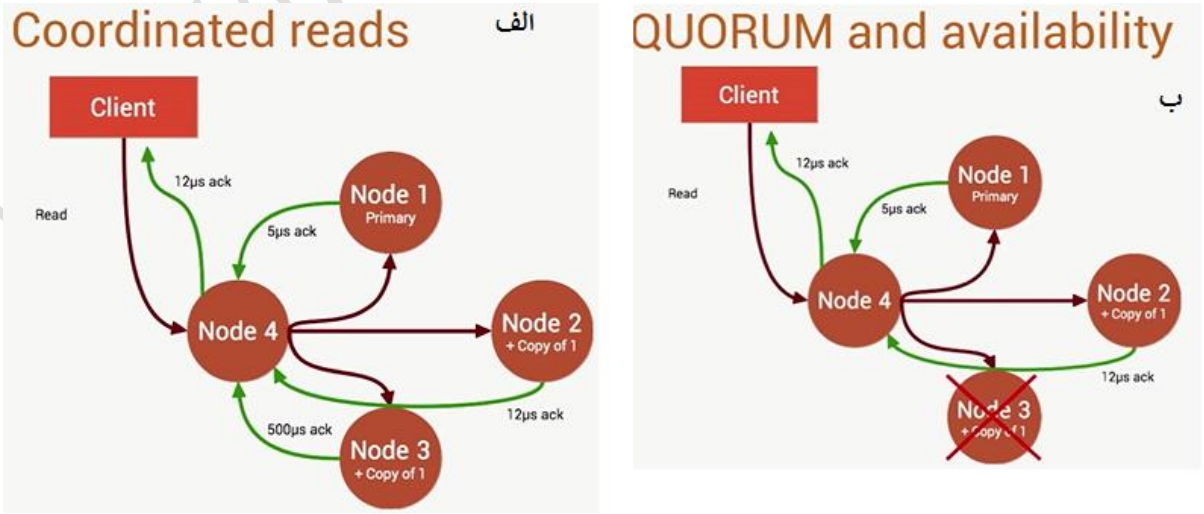
برای هر گره یک دامنه هش و برای هر داده بر اساس کلید یک هش نسبت داده می‌شود که این در شکل ۳-۳۳-الف نشان داده شده است. اما همانطور که می‌بینید Jim دارای هشی می‌باشد که رقم اول سمت چپ آن ۵ می‌باشد و در محدوده هش گره C قرار می‌گیرد بنابراین در گره C ذخیره خواهد شد. همچنین Carol دارای هش با رقم سمت چپ a می‌باشد که در محدوده هش گره D قرار خواهد گرفت و در آن نوشته خواهد شد. همینطور Johnny در گره A و suzy در گره C قرار خواهند گرفت. اما رپلیکاها چگونه قرار خواهند گرفت؟ با فرض Replication factor=3 که در شکل ۳-۳۳-ب نشان داده شده است و با توجه به اینکه Carol در گره D قرار می‌گیرد. این گره در جهت عقربه های ساعت دو کپی از آن را در گره‌های A,B ذخیره خواهد کرد. اما سطح Consistency چقدر خواهد بود؟ سازگاری برای هر عمل خواندن و نوشتن قابل تنظیم می‌باشد. اگر Replication factor=n باشد حداقل سازگاری زمانی بدست می‌آید که فاکتور سازگاری یک باشد یعنی برای دریافت Acknowledge منتظر پاسخ یک رپلیکا باشیم. اما برای سازگاری مطلوب باید یک حد نصاب (Quorom) در نظر گرفت. حدنصاب مطلوب این است که ۵۱٪ رپلیکاها عمل نوشتن را تایید کنند. اما Local-Quorom مطلوب زمانی است که ۵۱٪ رپلیکاهای محلی در یک دیتاسنتر تاییدیه بفرستند. برای سه عدد رپلیکا تعداد ۲ عدد تاییدیه مطلوب می‌باشد که در صورت تنظیم آن به عدد ۳ حداکثر سازگاری بدست می‌آید. اما سرعت و دسترس‌پذیری کم می‌شود.

در کاساندرا هم مثل داینامو از گره‌های مجازی استفاده می‌شود. این گره‌های مجازی در هنگام حذف و اضافه گره‌ها مفید واقع می‌شوند اما لزوما این گره‌های مجازی بطور متوالی مثل داینامو قرار ندارند. این روند در شکل ۳-۳۳-ج نشان داده شده است.



شکل ۳-۳۳: ساختار هش کردن، کپی و گره مجازی در کاساندرا

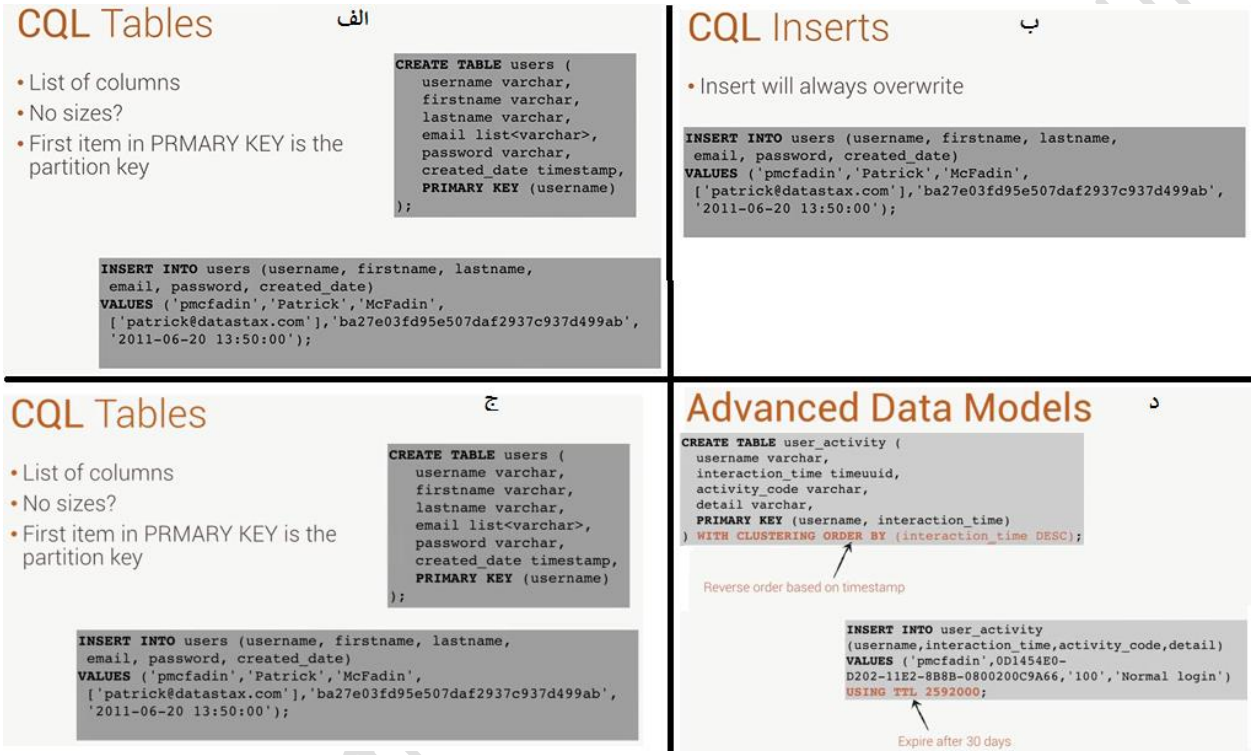
عمل خواندن در کاساندرا: در شکل ۳-۳۴-الف مشاهده می‌کنید که کلاینت درخواست خواندن به گره ۴ می‌دهد. گره ۴ متوجه می‌شود داده‌ها در گره‌های ۱-۲-۳ قرار دارند و درخواست را به آنها می‌فرستد. طبق شکل گره ۱ بعد از ۵ میکروثانیه، گره ۲ بعد از ۱۲ میکرو ثانیه و گره ۳ بعد از ۵۰۰ میکروثانیه به گره ۴ پاسخ می‌دهند؟ سوال اینجاست آیا گره ۴ منتظر پاسخ همه گره‌های فوق باشد؟ پاسخ وابسته به فاکتور خواندن است که در اینجا دو در نظر گرفته شده است و طبق شکل ۳-۳۴-ب پاسخ بعد از ۱۲ میکروثانیه به کلاینت داده می‌شود و پاسخ گره ۳ نادیده گرفته می‌شود.



شکل ۳-۳۴ عمل خواندن در کاساندرا

زبان پرس و جوی کاساندرا CQL=Cassandra Query Language

در شکل ۳-۳۵ CQL معرفی شده است. در شکل ۳-۳۵ الف ساختار کلی آن مطرح شده است. در شکل ۳-۳۵-ب دستور Insert مطرح شده است و نکته کلیدی رونویسی دستور است یعنی داده‌های مشابه قبلی با داده جدید جایگزین خواهند شد. در شکل ۳-۳۵-د شکل دستور ایجاد جدول پیشرفته را مشاهده می‌کنید که نکته کلیدی آن شرط ایجاد آن است که زمان انقضای جدول را هم مشخص می‌کند که در اولی براساس ترتیب خوشه بندی بصورت نزولی براساس زمان و در گزینه دوم براساس زمان چرخه عمر TTL معادل ۳۰ روز ایجاد شده است که بعد از ۳۰ روز جدول از بین خواهد رفت.



شکل ۳-۳۵: زبان پرس و جوی کاساندرا CQL

فصل چهارم: بسته های نرم افزاری و زبان‌های مرتبط با کلان داده

زبان SCALA: این زبان مخفف Scalable Language می‌باشد، و یک زبان برنامه نویسی عملکردی ترکیبی[1] است. این زبان توسط Martin Odersky ایجاد شد. اسکالا به نرمی ویژگی‌های زبان‌های شیء گرا و عملکردی را ترکیب کرده است. اسکالا برای اجرا روی ماشین مجازی جاوا JVM کامپایل می‌شود. شرکت‌های زیادی که کارهایشان وابسته به جاوا می‌باشد به اسکالا سوئیچ کرده‌اند تا قابلیت توسعه، مقیاس‌پذیری و قابلیت اطمینان کلی‌شان را بهبود دهند.

در اینجا بعضی نکات کلیدی را که باعث شده است اسکالا انتخاب اول توسعه دهندگان اپلیکیشن‌ها شود را بیان می‌کنیم.

اسکالا شیء گرا می‌باشد: اسکالا یک زبان شیء گرای خالص است که هر مقدار یک شیء می‌باشد. انواع و رفتار اشیاء با کلاس‌ها و ویژگی‌ها توصیف می‌شوند که در ادامه توضیح داده خواهند شد.

اسکالا عملکردی است: اسکالا همچنین یک زبان عملکردی است که هر تابع یک مقدار است و هر مقدار یک شیء است بنابراین در انتها هر تابع یک شیء می‌باشد.

انواع در اسکالا ایستا می‌باشند: اسکالا برخلاف سایر زبان‌هایی که انواع ایستا دارند مثل ...,C,Pascal, Rust نیازی ندارد که شما اطلاعات افزونه برای نوع فراهم کنید. شما در اکثر حالات اصلا نیازی ندارید نوع تعریف کنید و نیازی ندارید آن را تکرار کنید.

اسکالا روی JVM اجرا می‌شود: اسکالا به Java Byte Code ترجمه می‌شود و می‌تواند توسط دستور 'scala' اجرا شود. دستور scala شبیه دستور java می‌باشند که کد اسکالای کامپایل شده را اجرا می‌کند.

اسکالا می‌تواند کد جاوا را اجرا کند: اسکالا شما را قادر می‌کند تمامی کلاس‌های Java SDK را استفاده کنید و همچنین کلاس‌های خودتان را و پروژه‌های جاوای خودتان را.

اسکالا پردازش همروند و سنکرون را انجام می‌دهد: اسکالا به شما اجازه می‌دهد الگوهای کلی برنامه‌نویسی را به روشی موثر بیان کنید. آن تعداد خطوط کد را کاهش می‌دهد. آن به شما اجازه می‌دهد کدهایتان را به روشی immutable بنویسید، که اجرای همروند و موازی آن را تسهیل می‌کند.

اسکالا در مقایسه با جاوا: اسکالا ویژگی‌هایی دارد که کاملا با جاوا متفاوت است. بعضی از اینها عبارتند از: همه انواع شیء هستند، رابطه نوع، توابع تودرتو، توابع شیء هستند، پشتیبانی از زبان‌های DSL، ویژگی‌ها، Closureها، پشستیبانی همروندی.

پایه‌های اسکالا

اگر شما جاوا را به خوبی فهمیده‌اید آنگاه یادگیری اسکالا برای شما ساده خواهد بود. مهمترین تفاوت املایی بین اسکالا و جاوا این هست که ';' در انتهای خط اختیاری می‌باشد.

در یک برنامه اسکالا، مجموعه‌ای از اشیاء می‌باشد که با متدها یکدیگر را فراخوانی می‌کنند. در ادامه تفاوت مفاهیم پایه اسکالا را بیان می‌کنیم:

- شیء: اشیاء حالت و رفتار[2] دارند. یک شیء یک نمونه از یک کلاس می‌باشد.

[1] Hybrid functional language
[2] States and behaviors

- کلاس: یک کلاس برای تعریف رفتارها/حالات اشیاء تعریف می‌شود.
- متدها: یک متد اساساً یک رفتار می‌باشد. یک کلاس می‌تواند متدهای زیادی داشته باشد. در متدها منطق‌ها نوشته می‌شوند، داده‌ها دستکاری می‌شوند و همه عمل‌ها اجرا می‌شوند.
- فیلدها: هر شیء یکسری متغیر منحصر به خودش را دارد که فیلد نامیده می‌شوند. حالت یک شیء با مقادیر اختصاص داده شده به فیلدها ایجاد می‌شود.
- Closure: یک closure یک تابع است، مقدار بازگشتی آن وابسته به مقدار یک یا چند متغیر تعریف شده در خارج تابع می‌باشد.
- Traits: یک Trait تعریف متدها و فیلدها را در برمی‌گیرد، که با ترکیب آنها می‌توان آنها را در توابع استفاده کرد.

اولین برنامه اسکالا:

ما می‌توانیم یک برنامه اسکالا را در دو مد interactice و script mode اجرا کنیم.

مد محاوره ای: خط فرمان را باز کنید و دستور scala را در آن اجرا کنید و وارد خط فرمان اسکالا می‌شوید سپس در خط فرمان می‌توانید دستورات اسکالا را بنویسید مثلا دستور Scala>println(“Hello, scala”); پیام Hello Scala! را نشان خواهد داد.

مد اسکریپت: در این مد باید دستورات را در یک فایل بنویسید یا در محیط های اسکالا یا در محیطی مثل notepad.

```
object HelloWorld {
   /* This is my first java program.
    * This will print 'Hello World' as the output
    */
   def main(args: Array[String]) {
      println("Hello, world!") // prints Hello World
   }
}
```

مثلا با نوشتن کد فوق در notepad و ذخیره آن به نام HelloWorld.scala آنرا ابتدا با دستور scalac کامپایل کنید سپس با دستور scala اجرا کنید. بصورت زیر:

```
\>scalac HelloWorld.scala
\>scala HelloWorld
```

خروجی بصورت زیر خواهد بود:

```
Hello, World!
```

Basic Syntax:

در زیر syntax پایه‌ی برنامه‌نویسی اسکالا مطرح شده است.

- **حساس به حروف:** اسکالا به حروف حساس است یعنی شناسه Hello با hello فرق دارد و از دید اسکالا معانی متفاوت دارند.
- **نام کلاس‌ها:** برای نام کلاس، اولین حرف باید بزرگ باشد. اگر چندکلمه برای نامگذاری یک کلاس استفاده شوند حرف اول هر کلمه داخلی باید بزرگ باشد مثل: MyFirstScalaClass.
- **نام متد:** همه نام‌های متدها باید با حرف کوچک شروع شوند. اگر چند کلمه برای نام متد استفاده شوند حرف اول هر کلمه داخلی باید بزرگ باشند مثل: def myMethodName()
- **نام فایل برنامه:** نام برنامه باید دقیقاً با نام شیء تطابق داشته باشد. موقع ذخیره فایل باید نام شیء را استفاده کنید(دقت کنید اسکالا حساس به حروف است). پسوند فایل باید scala. باشد. مثلا فایل فوق با نام HelloWorld.scala ذخیره شد.
- **def main(args: Array[String]):** پردازش برنامه اسکالا از متد main شروع می‌شود. که بخش ثابت هر برنامه است.

شناسه‌های اسکالا: همه اجزاء اسکالا نیازمند نام هستند. اسامی برای اشیاء، کلاس‌ها، متغیرها و متدها و شناسه‌های فراخوان استفاده می‌شوند. یک keyword نمی‌تواند به عنوان اسم استفاده شود و شناسه‌ها حساس به حروف هستند. اسکالا چهار نوع شناسه را پشتیبانی می‌کند. ۱- شناسه‌های حرفی-عددی، ($رزرو شده است و نمی‌تواند به عنوان شناسه استفاده شود. ۲- شناسه‌های عملگر شامل #,~,?,:,+ که بصورت ترکیبی می توانند استفاده شوند مثل، <: ,<?> ,::: ,++ ,+ ۳-شناسه‌های ترکیبی مثل unary_+ و myvar_=. ۴- شناسه های literal که شامل رشته های دلخواه است که در یک '...' قرار می‌گیرد.

Scala Keywords: کلمات کلیدی اسکالا در زیر آمده است و نمی‌توانند به عنوان اسم دیگری استفاده شوند

abstract	case	catch	Class
def	do	else	extends
false	final	finally	For
forSome	if	implicit	import
lazy	match	new	Null
object	override	package	private
protected	return	sealed	super
this	throw	trait	Try
true	type	val	Var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

Spark: زیرساخت پردازش اطلاعات می‌باشد، یعنی چگونه داده‌های ذخیره شده را load کنیم و بصورت توزیع شده پردازش کنیم. یکی از مزایای Spark سرعت بالای آن می‌باشد مثلا در مقایسه با MapReduce ممکن است ده‌ها برابر سریعتر باشد. داده‌ها را در RAM پردازش می‌کند منتها پیچیدگی بیشتری دارد و ابزار خوبی برای تسریع پردازش می‌باشد.

What is Spark?

- Fast, expressive cluster computing system compatible with Apache Hadoop
  - Works with any Hadoop-supported storage system (HDFS, S3, Avro, …)
- Improves **efficiency** through:
  - In-memory computing primitives
  - General computation graphs

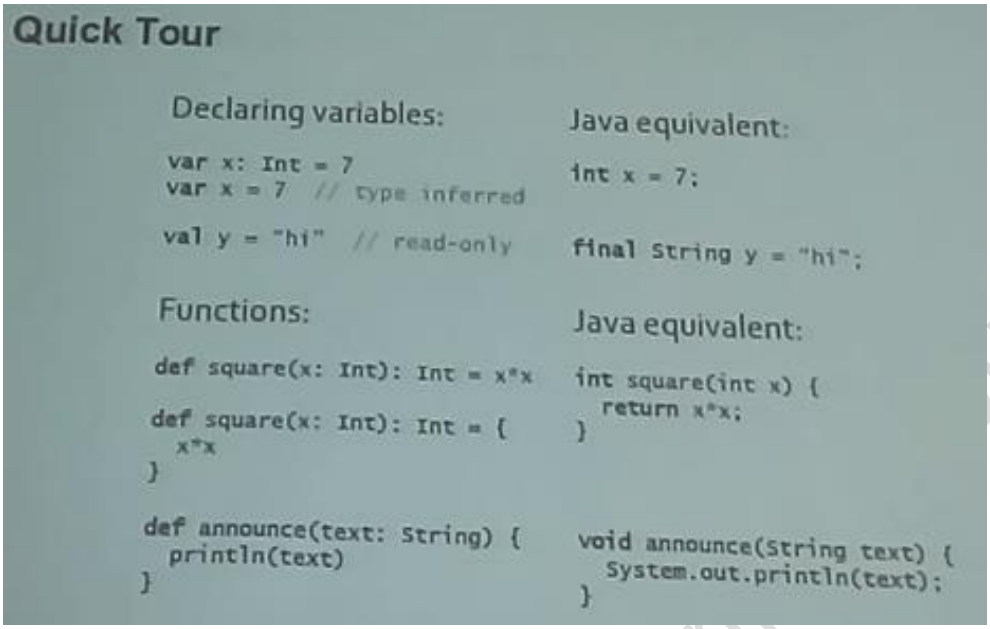
  → Up to 100× faster
- Improves **usability** through:
  - Rich APIs in Java, Scala, Python
  - Interactive shell

  → Often 2-10× less code

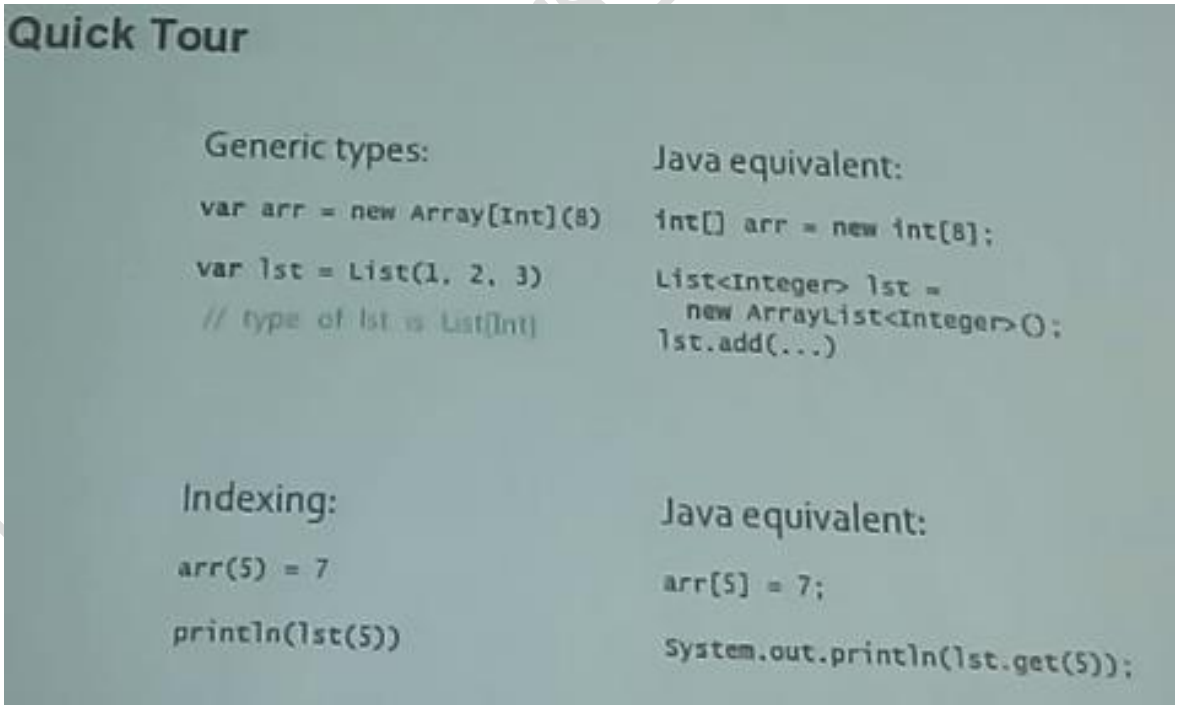
اسپارک را می توان در یک کامپیوتر یا در یک کلاستر یا یک کلود اجرا کرد. هم می توان در شل اسپارک کار کرد و هم بصورت محاوره ای آن را استفاده کرد.

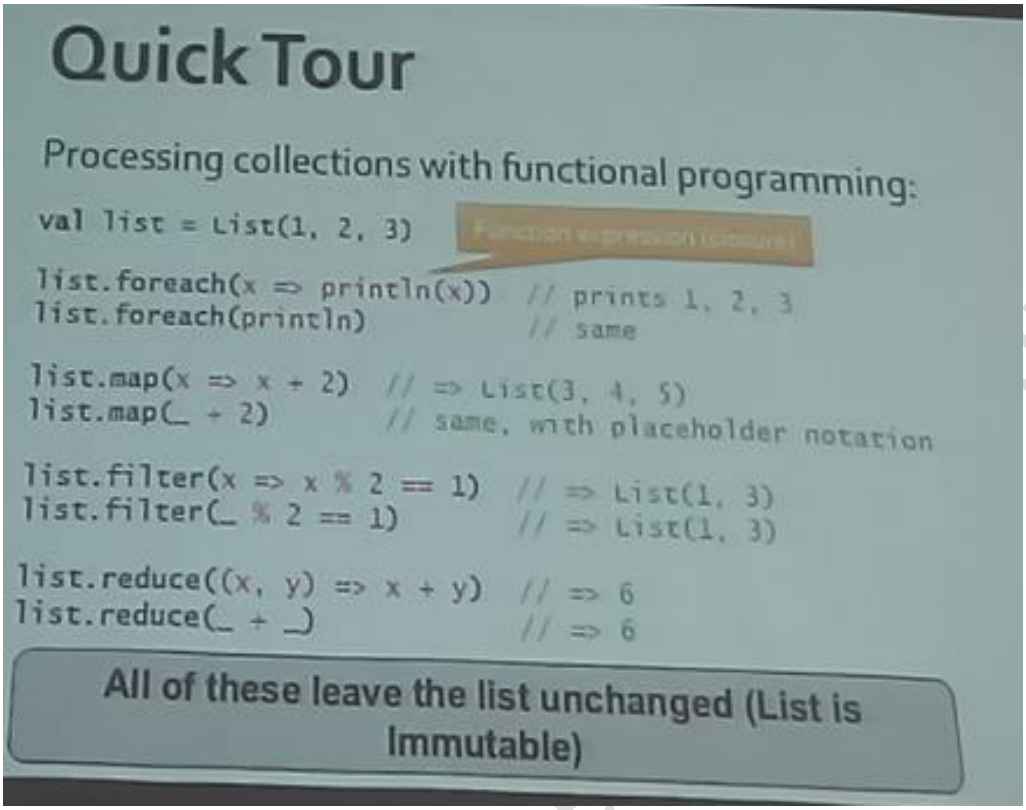
What is Scala?

High-level language for JVM
>> Object-oriented + Functional programming (FP)
Statically typed
>> Comparable in speed to Java
>> no need to write types due to type inference
Interoperates with Java
>> Can use any Java class, inherit from it, etc;
>> Can also call Scala code from Java



Immutable: نوع val غیر قابل تغییر است و read-only است این مزیت زیادی برای پردازش موازی ایجاد می‌کند. این منطق از اینجا نشات می گیرد که قرار نیست چیزی (رکوردی) از پایگاه حذف شود و اگر قرار هست مقدار جدیدی از رکورد نوشته شود با اسم دیگری نوشته خواهد شد. در بیگ دیتا بیشتر consistency , availability مهم است تا ظرفیت ذخیره سازی. بنابراین اشکالی ندارد چندین ورژن از یک مقدار را داشته باشیم. این خود به پردازش تاریخچه آن رکورد کمک می‌کند. این خود به توزیع مقادیر کمک می کند.

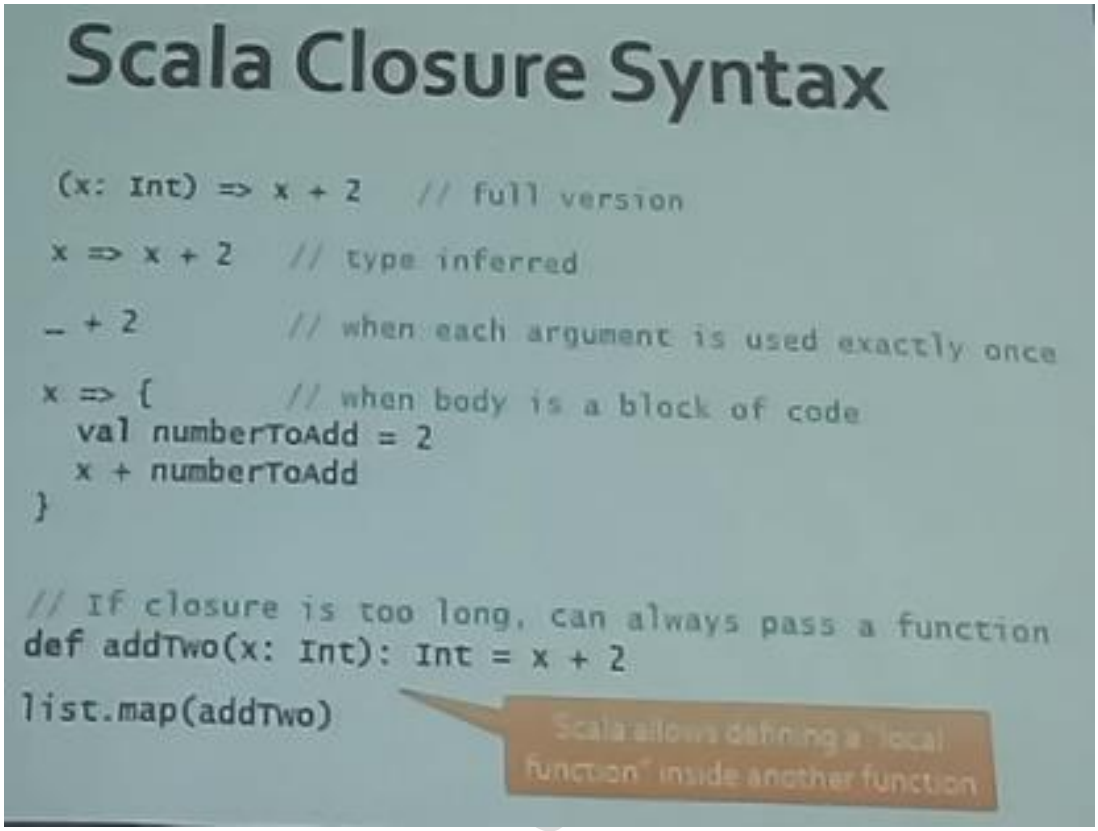




تفاوت For و foreach: در for ترتیب اجرا مهم هست ولی در foreach ترتیب اجرا مهم نیست و نسبت به اجرای آن بی تفاوت است. اگر روی یک نخ (thread) اجرا شود به ترتیب اجرا می شود ولی در چند گره ترتیب اجرا مهم نیست.

List.reduce((x,y)=>x+y) دو زوج (x,y) در صورت وجود با هم جمع شوند بدون توجه به ترتیب. این قضیه راجع به max, min درست هست. اما راجع به average درست نیست. چون در میانگین ترتیب مهم هست ولی در حداقل حداکثر مهم نیست. باید طوری کد نوشده شود که ترتیب مهم نباشد.

در شکل زیر همه موارد closure مثل هم هستند و شبیه f(x)=x+2 می‌باشد. در اسکالا اسم تابع و متغیر خیلی با هم تفاوت ندارد. حتی نوشتن اسم تابع زیاد مهم نیست. یعنی توابع بصورت بی نام هم تعریف می‌شود.



مثالی از استفاده از closure بصورت زیر است:

Scala Cheat Sheet

Variables:

```
var x: Int = 7
var x = 7     // type inferred

val y = "hi"  // read-only
```

Functions:

```
def square(x: Int): Int = x*x

def square(x: Int): Int = {
  x*x   // last line returned
}
```

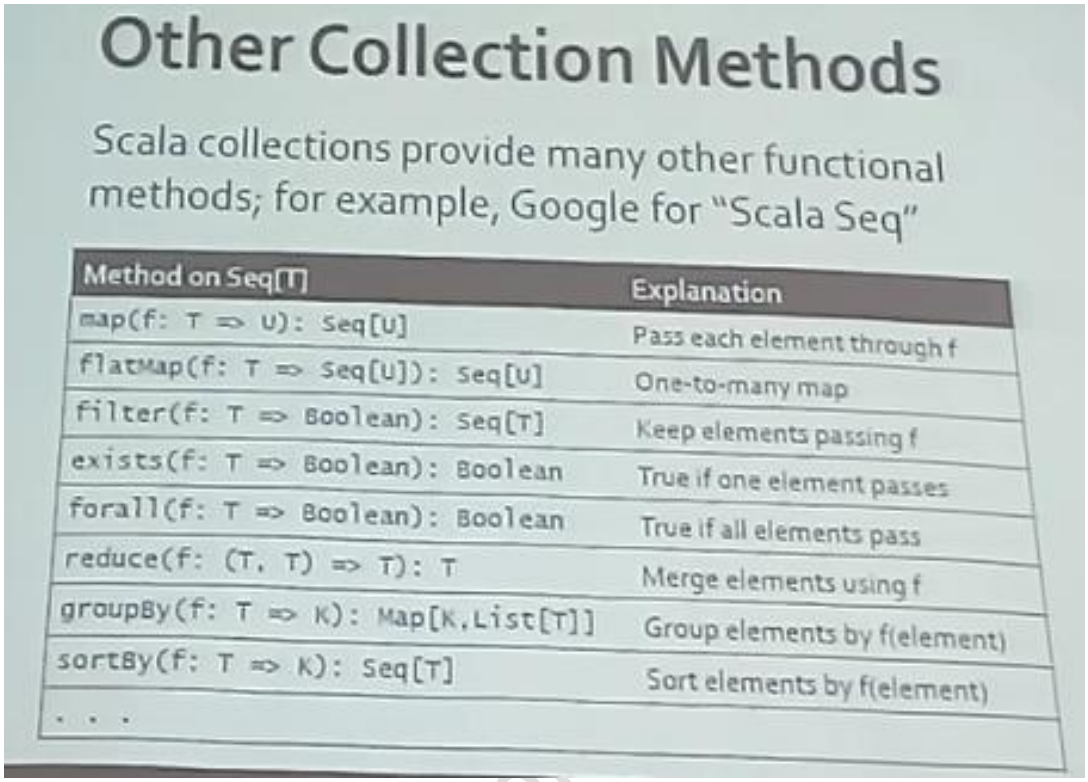
Collections and closures:

```
val nums = Array(1, 2, 3)

nums.map((x: Int) => x + 2) // => Array(3, 4, 5)

nums.map(x => x + 2)  // => same
nums.map(_ + 2)       // => same

nums.reduce((x, y) => x + y) // => 6
nums.reduce(_ + _)           // => 6
```

Java interop:

```
import java.net.URL

new URL("http://cnn.com").openStream()
```

More details:
scala-lang.org



flat map: one to many وMap: one to one .

Key Idea

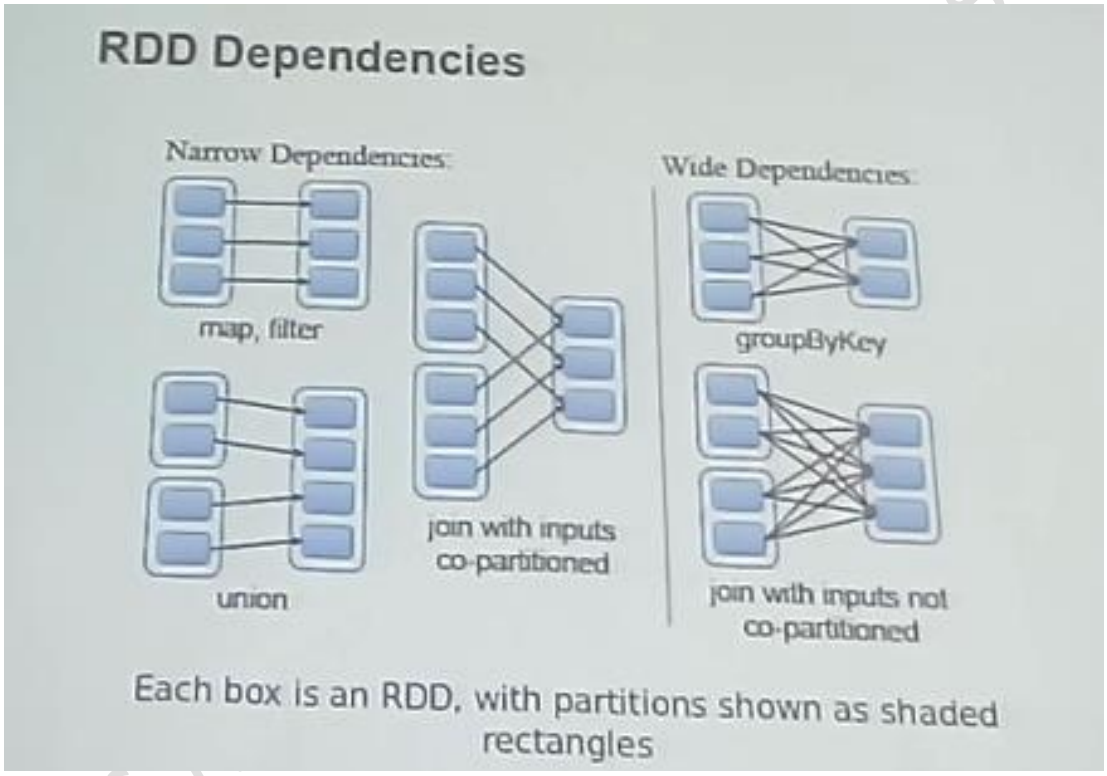
- **Work with distributed collections as you would with local ones**

- Concept: resilient distributed datasets (RDDs)
  - Immutable collections of objects spread across a cluster
  - Built through parallel transformations (map, filter, etc)
  - Automatically rebuilt on failure
  - Controllable persistence (e.g. caching in RAM)

اسپارک برای سریع کار کردن lazy کار می کند یعنی هر کار را در آخرین لحظه‌ای که لازم هست انجام می‌دهد. مثلاً اگر دو عدد را جمع کنیم سپس پرینت بگیریم تا زمان پرینت اعداد را جمع نمی‌کند. هدف این هست فقط جنبه هایی را که نیاز هست انجام دهد شاید بخش هایی نیاز به اجرا نداشته باشند.

RDD مجموعه داده‌ای است که هم توزیع شده است هم resilient هم immutable. چون immutable است به راحتی می‌توان آن را توزیع کرد. مسئله نسخه های مختلف داده‌ها وجود ندارد. Resilient یعنی چیزی که خراب نمی‌شود و اگر خراب شود دوباره می‌تواند درست شود. هر RDD یک شجره نامه دارد و معلوم هست که هر RDD از چه کدی تولید می‌شود و از چه RDDهای دیگری تولید شده است. بنابراین

در صورت گم شدن یک RDD می‌توان آن را از روی تاریخچه‌اش بازیابی کرد. RDD یک بسته اطلاعات میانی است که به سادگی قابل پخش کردن به سیستم‌های مختلف است. RDD ها به چندین Shard تقسیم می‌شوند. در شکل زیر وابستگی های بین RDD ها نشان داده شده است. در narrow dependency یک RDD که شامل سه shard است به یک RDD دیگر با سه shard دیگر map شده است. ورودی توزیع شده است خروجی هم توزیع شده است. هر کامپیوتری که یک shard را دارد map را به آن shard اعمال می‌کند و shard جدیدی تولید می‌شود که همه سیستم‌ها این کار را روی shardهای خودشان انجام داده اند و shard جدید را تولید کرده اند. که در مجموع یک RDDی توزیع شده دیگر تولید می‌شود. RDD یکسری بسته های اطلاعات توزیع شده هستند که روی کامپیوتر خاصی قرار ندارد بلکه روی یکسری کامپیوتر توزیع شده است و در RAM سیستم ها قرار دارد و قابل ذخیره دائمی هستند. RDDهای قبلی که محاسبه روی آنها انجام شده است سرجایشان هستند. در union دو RDD با هم یک RDD شده اند که shardها لزوماً در یک سیستم نیستند و توزیع شده اند. مبنای محاسبات در spark همین RDDها هستند.

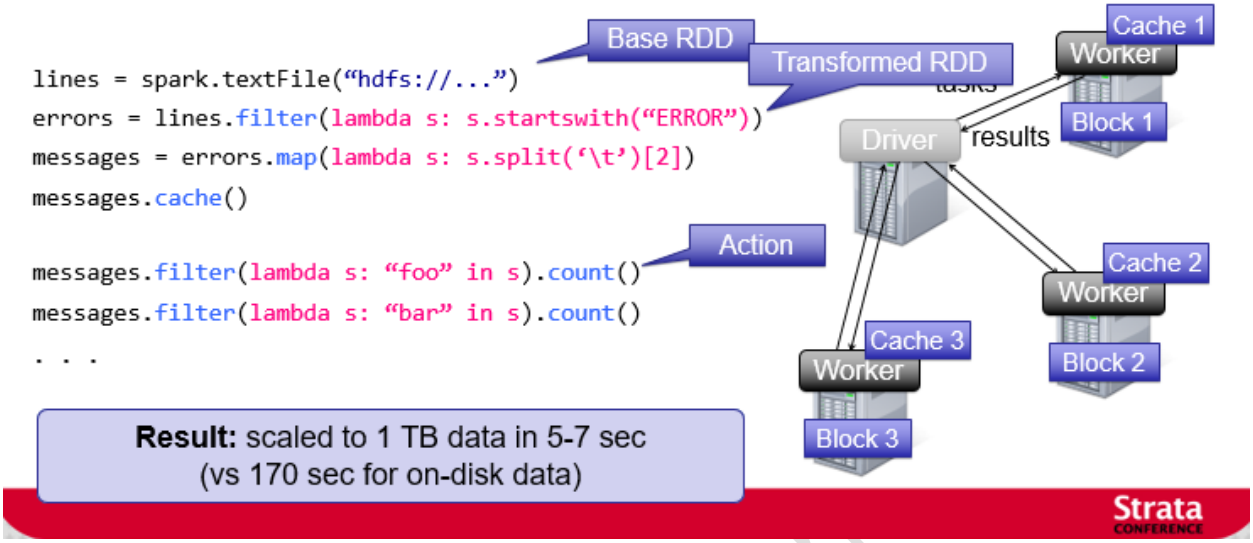


در شکل زیر مفهوم RDDها و lazy بودن نشان داده شده است.

Example: Mining Console Logs

- Load error messages from a log into memory, then interactively search for patterns

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(‘\t’)[2])
messages.cache()

messages.filter(lambda s: “foo” in s).count()
messages.filter(lambda s: “bar” in s).count()
. . .
```



در خط اول یک فایل به lines بار می شود. در اینجا lines خودش یک RDD خواهد یود که هر shard آن در یک کامپیوتر قرار دارد. (کد به زبان پایتون است). خطوط filter, map,cache هر کدام یک transform هستند. تا اینجا هنوز محاسبه ای انجام نمی‌شود. Count یک action است و در این خط عمل شروع می‌شود و عمل انجام می‌شود (lazy). عمل cache برای action count بعدی مناسب هست.

Operations

- Transformations (e.g. map, filter, groupBy, join)
  - Lazy operations to build RDDs from other RDDs
- Actions (e.g. count, collect, save)
  - Return a result or write it to storage

RDD Fault tolerance: مادامی که RDD های قبلی هستند RDDی حذف شده قابل بازیابی می‌باشد.

RDD Fault Tolerance

RDDs track the transformations used to build them (their *lineage*) to recompute lost data

E.g:

```
messages = textFile(...).filter(lambda s: s.contains("ERROR"))
                        .map(lambda s: s.split('\t')[2])
```

Creating RDDs

```
# Turn a local collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

```
val count = sc.parallelize(1 to NUM_SAMPLES).filter { _ =>
  val x = math.random
  val y = math.random
  x*x + y*y < 1
}.count()
println(s"Pi is roughly ${4.0 * count / NUM_SAMPLES}")
```

در ادامه توجه شما را به چند نمونه مثال جلب می‌کنیم:

در دستور زیر textFile یک RDD خواهد بود که فایل به آن بار می‌شود.

```
val textFile=sc.textFile("README.md")
```

دستور زیر خطوط فایل را در صفحه نمایش نشان خواهد داد.

```
textFile.collect().foreach(println)
```

دستور زیر اولین خط فایل را نمایش خواهد داد.

```
textFile.first()
```

دستور زیر با filter خطوطی را که شامل کلمه Spark هست را می شمارد.

```
textFile.filter(line=>line.contains("Spark")).count()
```

در دستور زیر خطوطی که شامل کلمه Spark هستند به linesWithSpark بار خواهند شد.

```
val linesWithSpark=textFile.filter(line=>line.contains("Spark"))
```

در دستور زیر خطوطی که در linesWithSpark هستند نشان داده خواهند شد.

```
linesWithSpark.collect().foreach(println)
```

دستور زیر تعداد کلمات سطری را برمی گرداند که بیشترین تعداد کلمه را دارد.

```
textFile.map(line=>line.split(" ").size).reduce((a,b)=>if (a>b) a else b)
```

دستور زیر تعداد کلمات فایل را بر می گرداند.

```
textFile.map(line=>line.split(" ").size).reduce((a,b)=>(a+b))
```

دستور زیر تعداد کلمات سطری را برمی گرداند که بیشترین تعداد کلمه را دارد.

```
textFile.map(line=>line.split(" ").size).reduce((a,b)=>Math.max(a,b))
```

در دستور زیر همه زوج مرتب های بصورت (word, count) در wordCounts قرار می‌گیرد.

```
val wordCounts=textFile.flatMap(line=>line.split(" ")).map(word=>(word,1)). reduceByKey
((a,b)=>a+b)
```

در دستور زیر همه زوج مرتب های بصورت (word, count) در wordCounts قرار می‌گیرد. و سپس نشان داده می‌شوند.

```
val wordCounts=textFile.flatMap(line=>line.split(" ")).map(word=>(word,1)). reduceByKey
((a,b)=>a+b).collect().foreach(println)
```

برای شمارش فقط یک کلمه خاص می توان از دستور زیر استفاده کرد.

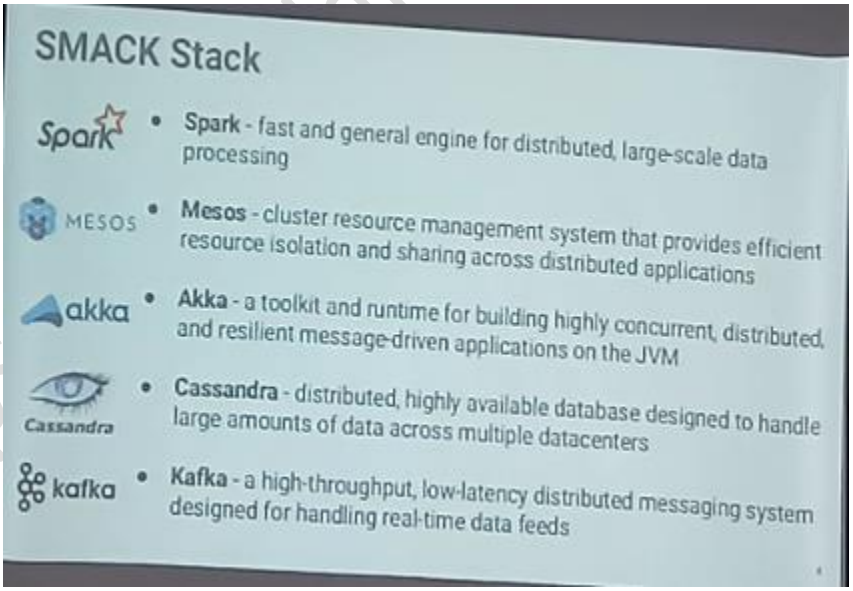
textFile.flatMap(line=>line.split(“ “)).filter(_==”myword”).map(word=>(word,1)). reduceByKey ((a,b)=>a+b).collect().foreach(println)

Scala Implementation

```
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
                  .mapValues(0.15 + 0.85 * _)
}

ranks.saveAsTextFile(...)
```

SMAK: یک stack برای استفاده اسپارک می‌باشد. در بحث کلان داده یک Stack دربرگیرنده لایه‌های زیرساخت می‌باشد و مجموعه‌ای از ابزارهای مورد نیاز برای کار با کلان داده و پردازش آن می‌باشد. مثلاً برای ذخیره سازی داده های از کاساندرا استفاده کنیم و برای پردازش داده‌ها از اسپارک استفاده کنیم. SMAK شامل Spark, Mesos, Akka, Cassandra, Kafka می‌باشد.

اسپارک یک موتور سریع و عمومی برای پردازش داده در مقیاس–بزرگ می‌باشد.



Mesos: یک سیستم مدیریت منابع کلاستر است که اشتراک و جداسازی منابع را به طور مفیدی در کاربردهای توزیع شده فراهم می‌کند. اگر بخواهیم در یک کلاستر چندین کامپیوتر را درگیر کنیم و پیکربندی و هماهنگ کنیم و مشخص کنیم کدام master یا slave است می توان از آن استفاده کرد.

Akka: یک تولکیت برای ساخت کاربردهای براساس–پیام کشسان، توزیع شده و همروند روی JVM می‌باشد. فرض کنید روی JVM می خواهید چند کد را بطور موازی روی کامپیوترهای مختلف اجرا کنید JVM ابزاری برای مدیریت این کدهای همروند ندارد و Akka می‌تواند این کار را انجام دهد و به مدیریت بهتر موازات و همروندی کمک کند.

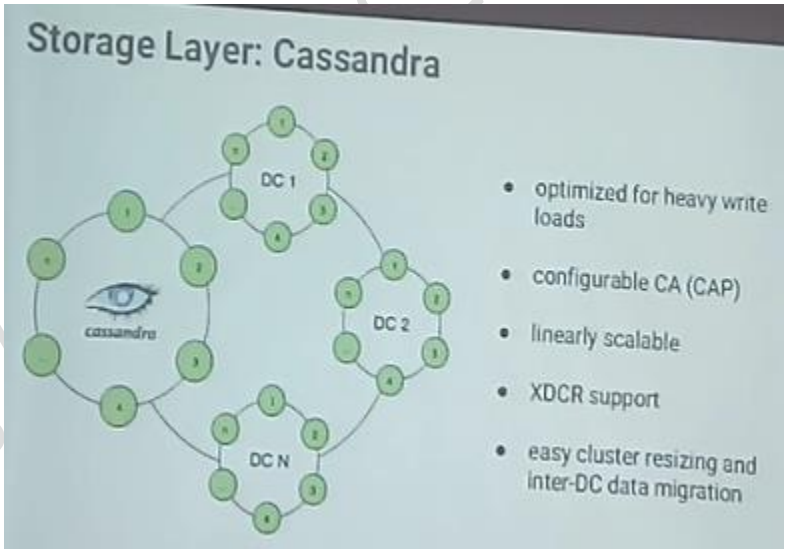
Cassandra: یک پایگاه داده توزیع شده با قابلیت دسترس بالا برای مدیریت میزان داده زیاد در چندین دیتاسنتر می‌باشد.

Kafka: یک سیستم پیام توزیع شده با توان عملیاتی بالا، تاخیر کم می‌باشد که برای مدیریت ورود داده‌های آنی طراحی شده است. مثلا در IOT سنسورهای بطور آنی و پیوسته جریانی از پیام ها را تولید می‌کنند که نیاز به پردازش دارند. این استریم از پیام ها توسط کافکا قابل مدیریت می‌باشد.

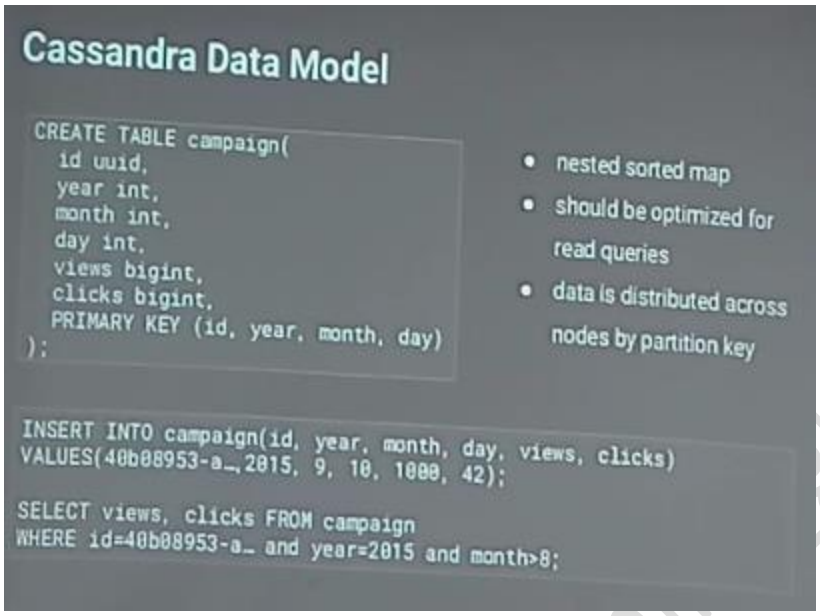
Pull: فرض کنیم می‌خواهیم از یک سنسور اطلاعاتی بخوانیم. اگر سرور به سنسور درخواست دهد که پیام را به او بفرستد و سنسور ارسال کند عمل pull انجام شده است.

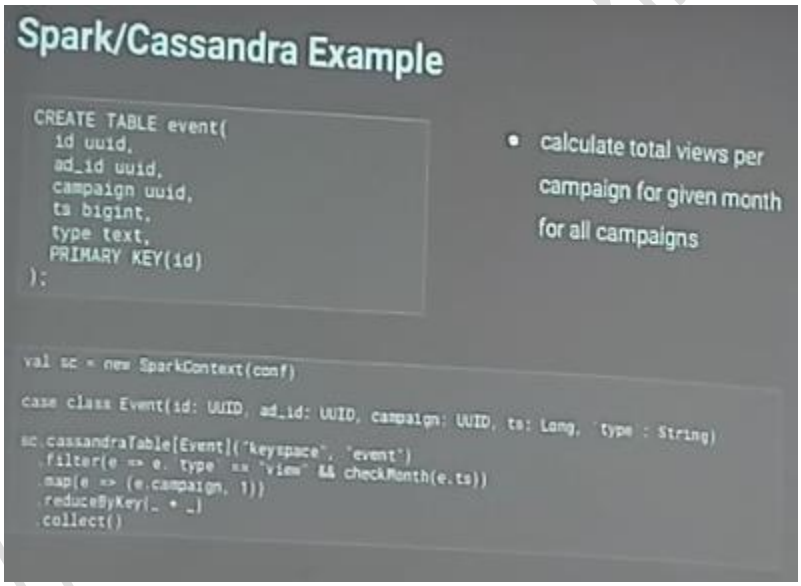
Push: در این حالت سرور به سنسور پیام نمی دهد که داده ها را ارسال کند بلکه خود سنسور داده ها را در کانال مد نظر را پوش می‌کند. بنابراین داده‌های زیادی از طریق تعداد زیادی سنسور ممکن است پوش شود و پردازش و مدیریت آنها سخت است. برای کار با این استریم های زیاد از یکسری ابزار استفاده می‌شود که آنها را دریافت و مدیریت کند و نیازی نیست کاربر بصورت دستی آنها را مدیریت کند. کافکا سیستمی است که می تواند این عمل را انجام دهد.

نکته: استفاده از یک استک براساس نیاز هست. یعنی ممکن است برای یک کاربرد خاص SMAK مفید باشد و برای کاربرد دیگری معماری دیگری با ابزارهای دیگری بهتر باشند.

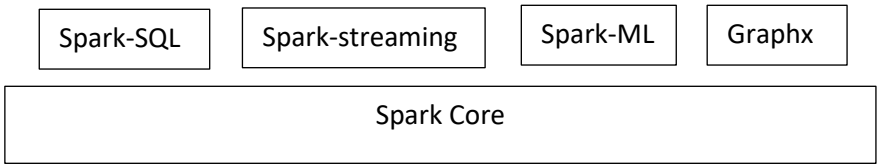


کاساندرا: لایه ذخیره سازی است که با قابلیت دسترس پذیری بالا می‌باشد و در فصل قبل به اختصار توضیح داده شد. همانطور که گفتیم کاساندرا براساس خانواده ستون می‌باشد و می‌توان برای آن keyspace تعریف کرد و در آن یکسری جدول ایجاد کرد. قبلا دستورات ایجاد جدول، درج و جستجوی داده‌ها در کاساندرا را توضیح دادیم به عنوان مثال در شکل زیر از این دستورات استفاده شده است. همانطور که قبلا گفتیم کاسندرا از CQL=Cassandra Query language استفاده می‌کند.





مزیت SMAK استفاده همزمان از قابلیت ها و توانایی‌های Spark/Cassandra می باشد. در شکل فوق جدول ایجاد شده در کاساندار با spark context با هم بطور همزمان استفاده شده است و اعمال map/reduce روی آنها انجام شده است. این قابلیت یعنی استفاده از توان عملیاتی بالای پردازش داده اسپارک و قابلیت ذخیره‌سازی بالای کاساندرا بطور همزمان. مثال فوق Campaign هایی که دارای رخداد view هستند را می‌شمارد. مزیت کد فوق این هست که کد کاملاً functional است و اگر مثلا جدول در ۱۰۰ کامپیوتر توزیع شده باشد هر کامپیوتر روی شارد خودش این عمل را انجام می‌دهد. در این مثال query در سطح پایین انجام می شود و در عین حال توزیع شده است و نیازی نیست داده ها مثل پایگاه داده‌های رابطه ای قدیمی روی یک کامپیوتر قرار داشته باشند. اسپارک بصورت یک core می‌باشد و چند ابزار می‌توانند روی آن اجرا شوند بصورت شکل زیر:



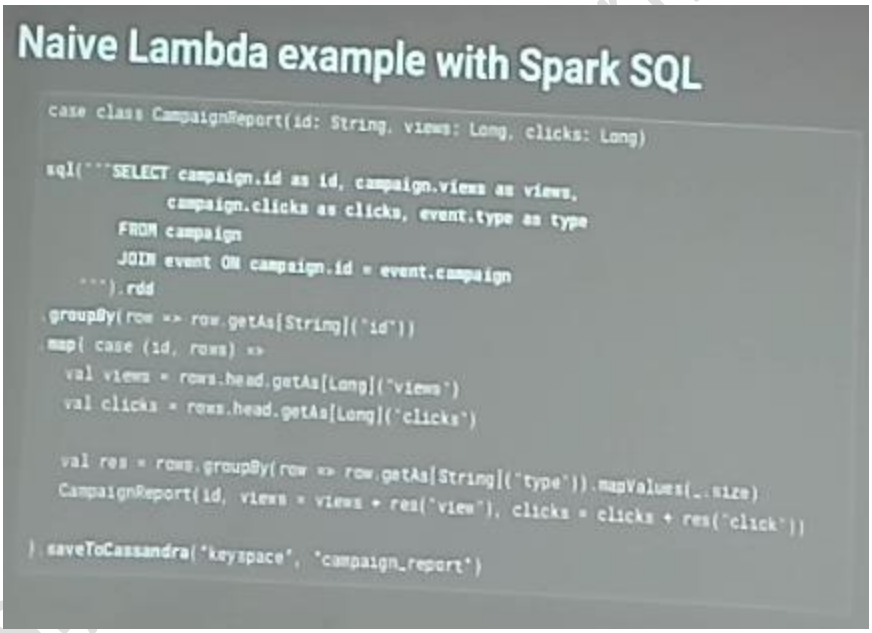
Spark-SQL: به شما کمک می‌کند با اسپارک پرس و جوهای SQL بزنید. تمام اتفاقات در spark-SQL ابتدا به زبان اسپارک ترجمه می‌شوند سپس اجرا می‌شوند. مثل استفاده از Hive , Pig در هدوپ.

Spark-Streaming: بعضی وقت‌ها داده‌ها بصورت batch است و بصورت انبوه از قبل وجود دارد و می‌خواهیم روی آنها پردازش انجام دهیم. اما گاهی داده‌ها بصورت استریم هستند و بصورت آنی وارد شده و باید پردازش شوند مثل داده‌های ورودی از سنسورهای IOT. در این نوع داده‌ها می‌توان از اسپارک–استریمینگ استفاده کرد.

Spark-ML: که شامل ابزارهای یادگیری ماشین machine learning است و کارهایی مثل deep learning را می تواند انجام دهد.

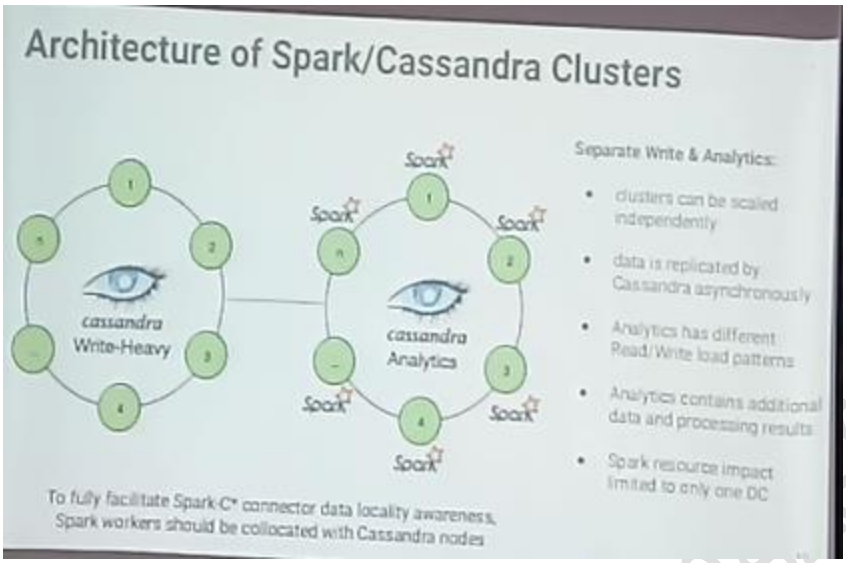
Graphx: بعضی مواقع داده‌ها بصورت گراف هستند مثل داده‌های مربوط به شبکه‌های اجتماعی (یا بعضی داده‌های GIS) و برای پردازش نیازمند در نظر گرفتن ارتباطات بین یال‌ها و گره‌ها می‌باشند. مثلاً اینکه یک فرد با چند نفر دوست هست و چه ارتباطاتی دارد. برای این نوع داده ها می‌توان از GraphX استفاده کرد. اگر این داده ها را بصورت SQL یا جدولی ذخیره کنیم آنقدر تعداد گره ها و یال‌ها زیاد هست که زمان پاسخ آن خوب نخواهد بود. در این حالات باید از ابزارهای منطبق با گراف استفاده کرد.

در شکل زیر نمونه ای از کد نوشته شده با Spark-SQL را مشاهده می‌کنید.

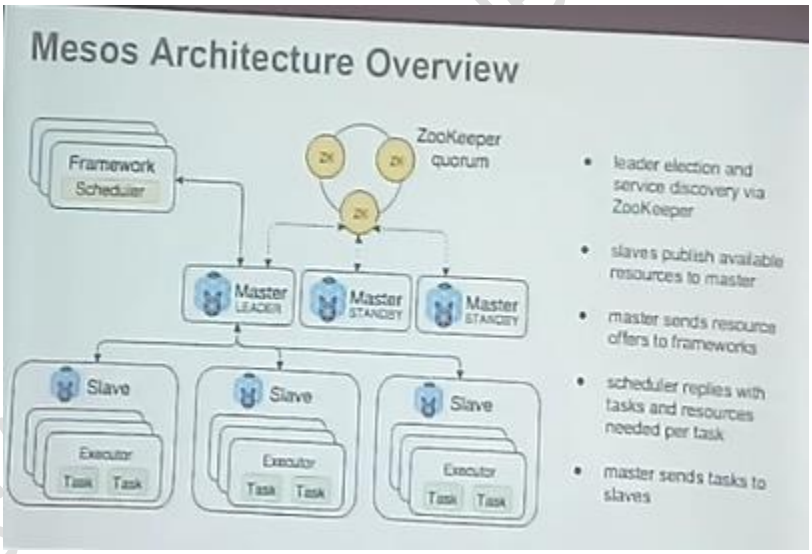


در این کد مشاهده می شود که در ابتدا از دستور SQL استفاده شده است و بعد از آن از دستورات اسپارک map و .. استفاده شده است. این ابزار بسیار ابزار مفیدی است که می تواند هم بصورت توزیع شده اجرا شود و هم پرس و جوهای SQL را انجام دهد. برای نوشتن در این ابزار هم باید Fuanctional فکر کرد، هم بصورت توزیع شده فکر کرد و هم از پرس و جوهای SQL استفاده کرد.

در معماری Spark-Cassandra دو حلقه تعریف می‌شود که در شکل زیر نشان داده شده است. در حلقه اول سمت راست که به آن Cassandra-analytics می‌گویند گره‌ها هم اسپارک هم کاساندار دارند دستورات اسپارک–کاساندرا روی این گره‌ها اجرا می‌شوند. حلقه سمت چپ که به آن Cassandra write-heavy می‌گویند، گره ها فقط کاساندرا دارند و مکانیزم ذخیره سازی را ایجاد می‌کنند.



معماری Mesos: یک سیستمی است که روی تعدای سیستم روی کلاستر نصب می شود و مشخص می کنیم که کدام سیستم کلاینت و کدام سرور است. معماری آن در شکل زیر نشان داده شده است. Zookeeper برای انتخاب سرور از اجماع استفاده می کند و مکانیزمی جدا از Mesos دارد.

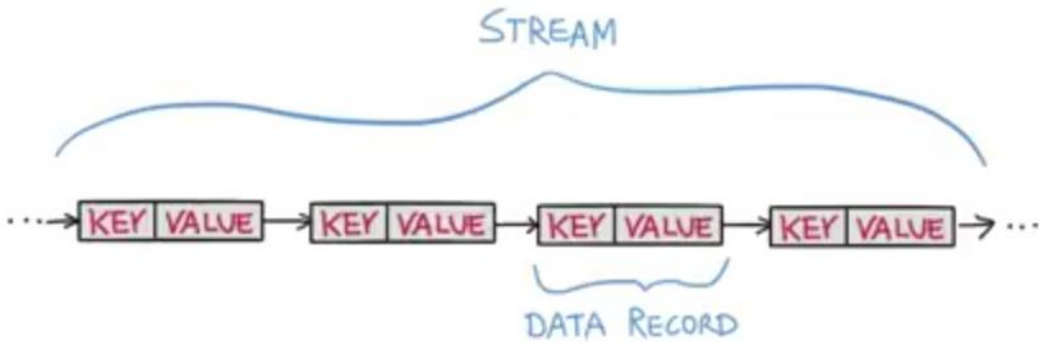


کافکا یک ابزار پردازش استریم می‌باشد که این استریم ها را می‌تواند به اسپارک بدهد.

استریم چیست؟ جریانی آنی، پیوسته و بدون کران از رکوردها می‌باشد. شما نیازی ندارید که صریحاً رکوردهای جدید را درخواست دهید شما آنها را دریافت می‌کنید. رکوردها زوج های key-value می‌باشند.

What is a Stream?

- **Think of a stream as an unbounded, continuous real-time flow of records**
  - You don't need to explicitly request new records, you just receive them
- **Records are key-value pairs**

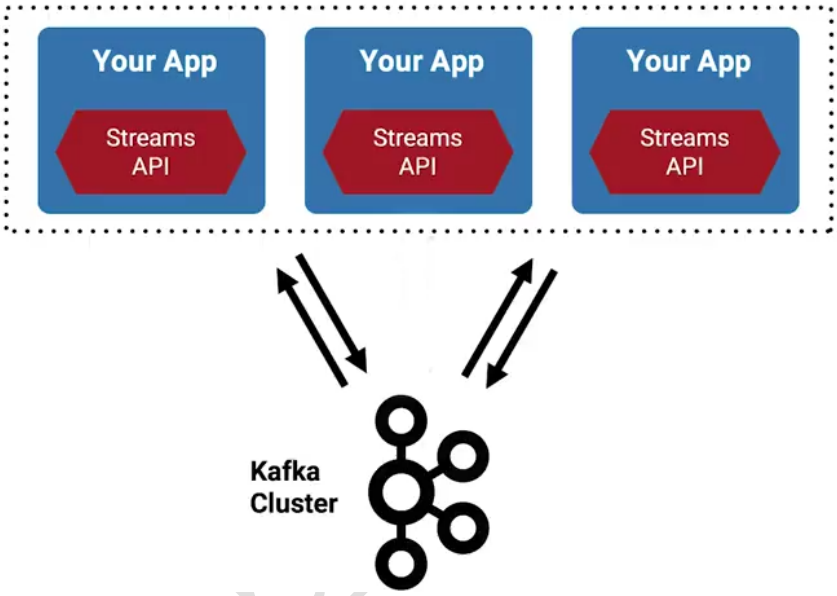


What Is the Kafka Streams API?

- **The Kafka Streams API transforms and enriches data**
  - Supports per-record stream processing with millisecond latency
  - Supports stateless processing, stateful processing, windowing operations
- **Write standard Java applications and microservices to process your data in real time**
  - No separate processing cluster required
  - Develop on Mac, Linux, Windows
  - Elastic, highly scalable, fault-tolerant
  - Deploy to containers, VMs, bare metal, cloud, on prem
  - Equally viable for small, medium, and large use cases
  - Fully integrated with Kafka security
  - Supports exactly-once semantics as of 0.11.0
- **The Kafka Streams API is part of the open-source Apache Kafka project**

Using the Kafka Streams API

- **Call the Kafka Streams API from your Java or Scala applications**
  - The Kafka Streams API interacts with a Kafka cluster
  - The application does not run directly on Kafka brokers

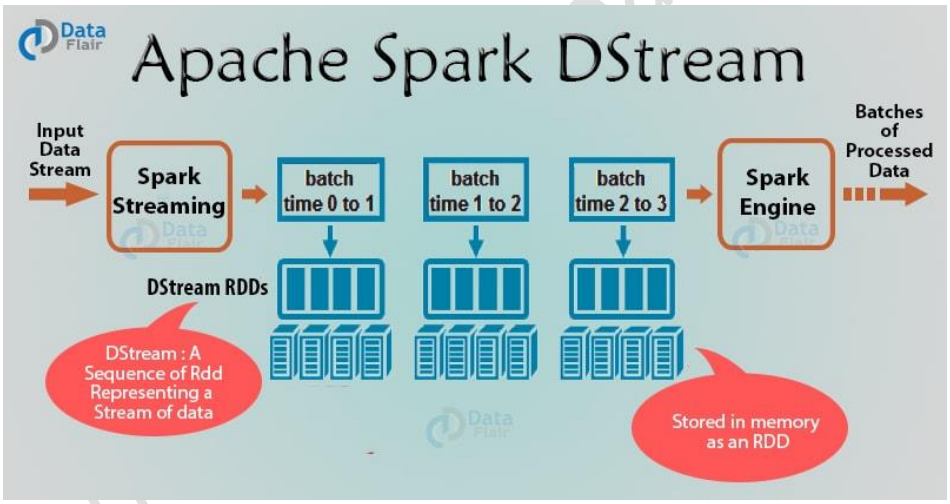


Spark-streaming: گاهی اوقات نیاز به پردازش های استریمینگ داریم. مثلا ممکن است سروری داشته باشیم که بطور پیوسته logهایی را تولید کند و ما بخواهیم بطور پیوسته این log ها را پردازش کنیم. یا مثلا بصورت پیوسته اخبار داخل سایت را به محض ورود پردازش کنیم یا داده‌های سنسورها را به محض ورود پردازش کنیم. هرچه IOT بیشتر پیشرفت کند میزان پردازش استریمینگ مهمتر خواهد شد.

در استریمینگ یکی از مواردی که باید مد نظر قرار گرفت حدمیانه ای بین latency, throughput می باشد. که براساس کاربرد متفاوت است و می توان آن را تنظیم کرد. اسپارک استریمینگ، استریم را قطعه قطعه می کند و به اسپارک می‌دهد و پردازش می‌کند. همانطور که گفتیم استریمینگ رشته ای بدون کران از داده‌هایی است که بطور پیوسته می‌رسند. استریمینگ جریان داده پیوسته را به واحدهای گسسته (discrete) برای پردازش بیشتر می‌شکند. پردازش استریم یک پردازش با تاخیر کم و آنالیز داده‌های پیوسته می‌باشد. اسپارک استریمینگ روی هسته اسپارک ۲۰۱۳ اضافه شد که یک پردازش مقیاس پذیر، با توان عملیاتی بالا و تحمل پذیر خطا برای داده‌های استریم تهیه می‌کند. ورود داده (data ingestion) می‌تواند در منابع مختلفی انجام شود مثل kafka, Apache Flume, Amazon kinesis or TCP sockets و پردازش

می‌تواند با استفاده از الگوریتم‌های پیچیده انجام شود که با توابع سطح بالای map, reduce,join, window بیان می‌شوند. در انتها داده‌های پردازش شده می‌توانند به سیستم فایل ، پایگاه‌های داده یا دشبوردهای زنده پوش شوند. عمل داخلی آن بصورت شکل زیر انجام می‌شود. داده‌های ورودی زنده، دریافت می‌شوند و توسط spark streaming به batchهایی تقسیم می‌شوند این batchها توسط موتور spark پردازش می‌شوند تا جریان نهایی نتایج را بصورت batch تولید کنند.

کلید استریمینگ در آپاچی اسپارک Discretized Stream یا Spark DStream می‌باشد که نشان دهنده جریانی از داده‌های می‌باشد که به batchهای کوچک تقسیم شده‌اند. Dstreamها روی Spark RDD ساخته شده‌اند. که باعث می‌شود بتوان از اجزای دیگر اسپارک مثل MLlib و Spark SQL برای آن استفاده کرد. Dstream جریانی پیوسته از داده‌هایی است که ورودیش را از منابع مختلفی مثل Kafka, Flume, Kinesis, or TCP sockets دریافت می‌کند. Dstream جریانی پیوسته از RDDها می‌باشد. هر RDD در Dstream شامل داده‌هایی از بازه‌های زمانی خاصی می‌باشد. هر عملی که روی یک DStream انجام می‌شود روی همه RDD های تحتانیش نیز اعمال می‌شود.



مثال: در زیر یک برنامه نمونه را می‌آوریم که تعداد کلمات را از یک متن دریافتی از یک سرور داده که به TCP socket گوش می‌کند می‌شمرد.

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3

// Create a local StreamingContext with two working thread and batch interval of 1 second.
// The master requires 2 cores to prevent from a starvation scenario.

val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

با این محتوا، می‌توانیم یک DStream ایجاد کنیم که نشان دهنده داده‌های استریمینگ از یک منبع TCP با نام میزبان (localhost) با پورت (۹۹۹۹) می‌باشد.

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

Lines DStream نشان دهنده جریانی از داده می‌باشد که از سرور داده دریافت می‌شود. هر رکورد در این DStream یک خط متن می‌باشد. سپس می‌خواهیم خطوط را با space به کلمات بشکنیم.

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

Flatmap یک عمل DStream یک به چند می‌باشد که یک DStream جدید با ایجاد چندین رکورد جدید از رکورد منبع DStream تولید می‌کند. در این حالت هر خط به چندین کلمه در جریان کلمات شکسته می‌شود که با words DStream نشان داده شده است. سپس می‌خواهیم تعداد کلمات را بشماریم.

```
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()
```

words DStream به یک DStream زوج (word,1)، map می‌شود، که سپس reduce می‌شود تا فرکانس تکرار کلمات را در هر داده batch بدست آورد. سپس wordCounts.print() تعداد کلماتی که در هر ثانیه تولید می‌شوند را چاپ می‌کند.

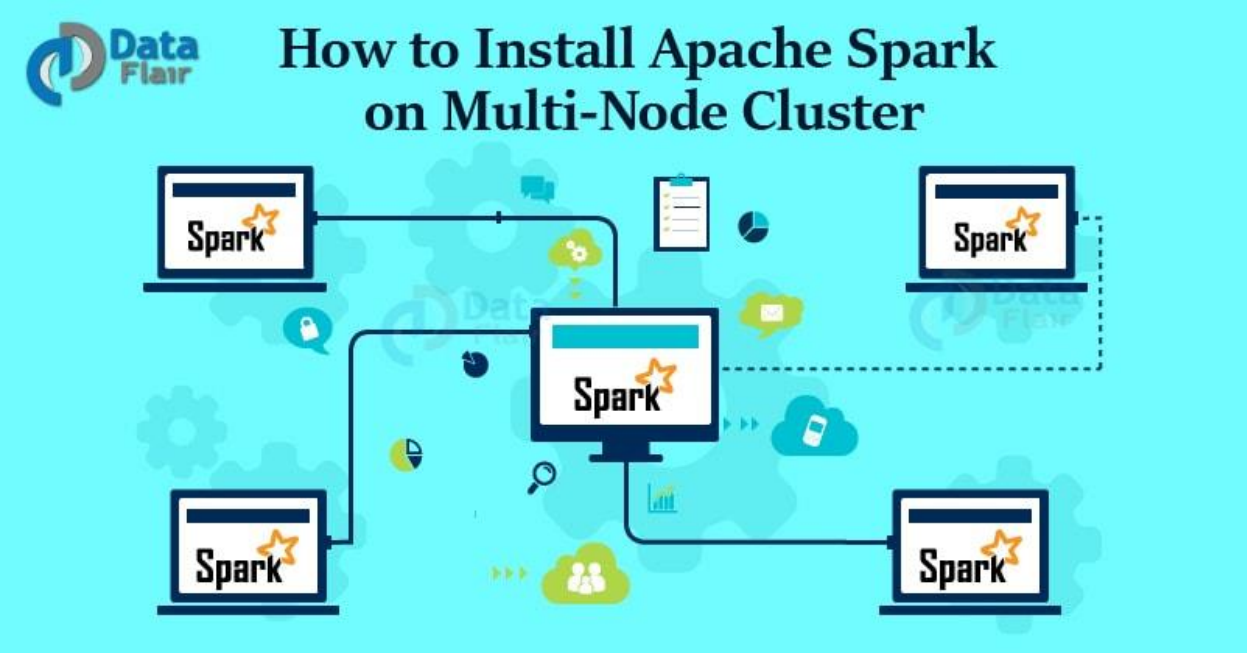
نکته اینکه وقتی این خطوط اجرا می‌شوند، Spark Streaming محاسبات را تنظیم می‌کند و هنوز پردازش واقعی شروع نشده است. برای شروع پردازش بعد از اینکه همه transformationها تنظیم شدند دستورات زیر را فراخوانی می‌کنیم:

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

کد کامل این برنامه بصورت زیر است:

```
package org.apache.spark.examples.streaming
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
object NetworkWordCount {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: NetworkWordCount <hostname> <port>")
      System.exit(1)
    }
    StreamingExamples.setStreamingLogLevels()
    // Create the context with a 1 second batch size
    val sparkConf = new SparkConf().setAppName("NetworkWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(1))
    // Create a socket stream on target ip:port and count the
    // words in input stream of \n delimited text (eg. generated by 'nc')
    // Note that no duplication in storage level only for running locally.
    // Replication necessary in distributed scenario for fault tolerance.
    val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
// scalastyle:on println
```

چگونگی نصب اسپارک در یک کلاستر با چند گره: در اینجا دستورات گام به گام برای بکارگیری و پیکربندی Apache Spark رو یک کلاستر چند گرهه واقعی را توضیح می‌دهیم. وقتی نصب تمام شد شما می‌توانید با اسپارک کار کنید و داده‌ها را پردازش کنید.



برای نصب اسپارک روی یک کلاستر چندگرهه مراحل زیر را دنبال کنید:

۱- پلت فرم‌های توصیه شده:

- سیستم عامل: لینوکس به عنوان یک پلت فرم توسعه و کاربرد پشتیبانی می‌شود. شما می‌توانید از Ubunto 14.04/16.04 یا نسخه‌های بعدی استفاده کنید. (شما می توایند از بقیه لینوکس ها مثل CentOS, Redhat و غیره نیز استفاده کنید). ویندوز هم به عنوان dev platform پشتیبانی می‌شود.
- Spark: Apache Spark 2.x

برای نصب Apache Spark روی یک خوشه چند گرهه، ما نیازمند چندین گره هستیم، یا شما می‌توانید Amazon AWS را استفاده کنید یا از راهنماهای نصب روی پلت فرم های مجازی مثل VMWare کمک بگیرید.

۲- نصب اسپارک روی Master

- پیش‌نیازها: همه ورودی ها را ویرایش کنید

Sudo nano /etc/hosts

سپس ورودی‌های master, Slave را وارد کنید.

```
MASTER-IP master
SLAVE01-IP slave01
SLAVE02-IP slave02
```

- جاوا ۷ را نصب کنید (Oracle Java توصیه می‌شود).

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java7-installer
```

- اسکالا را نصب کنید

```
sudo apt-get install scala
```

- SSH را پیکربندی کنید:

ابتدا SSH Server-Client را باز کنید

```
sudo apt-get install openssh-server openssh-client
```

سپس زوج کلید را تولید کنید.

```
ssh-keygen -t rsa -P ""
```

SSH بدون رمز را پیکر بندی کنید. محتوای ssh/id_rsa.pub. (از master) را به ssh/authorized_keys. کپی کنید.

همه slave ها را با SSH چک کنید.

```
ssh slave01
ssh slave02
```

- نصب اسپارک شما می‌توانید آخرین نسخه اسپارک را از http://spark.apache.org/downloads.html. دانلود کنید.. سپس آن را باز کنید tar xzf spark-2.0.0-bin-hadoop2.6.tgz
- سپس پیکربندی را تنظیم کنید. bashrc. را ویرایش کنید.

```
export JAVA_HOME=<path-of-Java-installation> (eg: /usr/lib
/jvm/java-7-oracle/)
export SPARK_HOME=<path-to-the-root-of-your-spark-
installation> (eg: /home/dataflair/spark-2.0.0-bin-
hadoop2.6/)
export PATH=$PATH:$SPARK_HOME/bin
```

Spark-env.sh را ویرایش کنید. ($spark_hone/conf) و پارامترهای زیر را تنظیم کنید:

یک کپی از spark-env.sh تهیه و آن را تغییر دهید.

```
cp spark-env.sh.template spark-env.sh
```

```
export JAVA_HOME=<path-of-Java-installation> (eg: /usr/lib
/jvm/java-7-oracle/)
export SPARK_WORKER_CORES=8
```

Slaveها را اضافه کنید: فایل پیکربندی slaveها را ایجاد کنید ($Spark_home/conf/) و ورودی‌های زیر را اضافه کنید:

```
slave01
slave02
```

اکنون Apache Spark با موفقیت در master نصب شده است، حال اسپارک را در همه slaveها بکار ببرید.

۳- نصب اسپارک در Slaveها

- پیش‌نیاز های نصب در همه Slaveها (یا گره های worker)

- “1.1. Add Entries in hosts file”
- “1.2. Install Java 7”
- “1.3. Install Scala”

  - نصب‌ها را از master به همه slaveها کپی کنید

a. Create tarball of configured setup

```
tar czf spark.tar.gz spark-2.0.0-bin-hadoop2.6
```

NOTE: Run this command on Master

b. Copy the configured tarball on all the slaves

```
scp spark.tar.gz slave01:~
```

NOTE: Run this command on Master

```
scp spark.tar.gz slave02:~
```

NOTE: Run this command on Master

نصب های پیکربندی شده در همه slaveها را un-tar کنید

```
tar xzf spark.tar.gz
```

حال apache Spark در همه slaveها نصب شده است. حال daemons را در کلاستر شروع کنید. کلاستر اسپارک را شروع کنید

I. Start Spark Services

```
sbin/start-all.sh
```

Note: Run this command on Master

II. Check whether services have been started

a. Check daemons on Master

```
jps
Master
```

b. Check daemons on Slaves

```
jps
Worker
```

Spark Web UI

I. Spark Master UI

Browse the Spark UI to know about worker nodes, running application, cluster resources.

http://MASTER-IP:8080/

II. Spark application UI

http://MASTER-IP:4040/

2.6. Stop the Cluster

I. Stop Spark Services

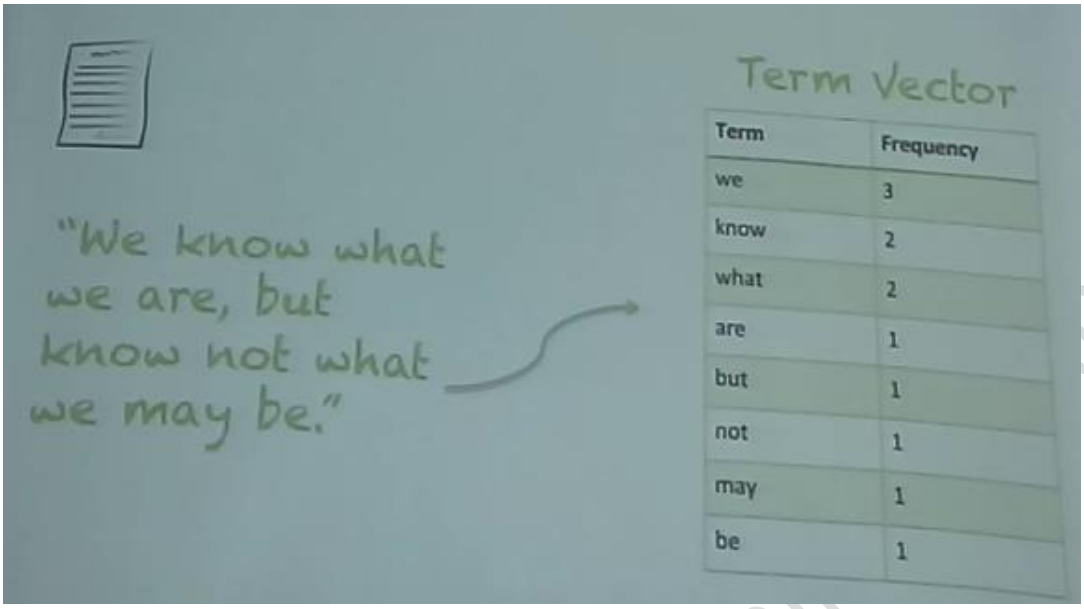
Once all the applications have finished, you can stop the spark services (master and slaves daemons) running on the cluster

```
1
   sbin/stop-all.sh
```

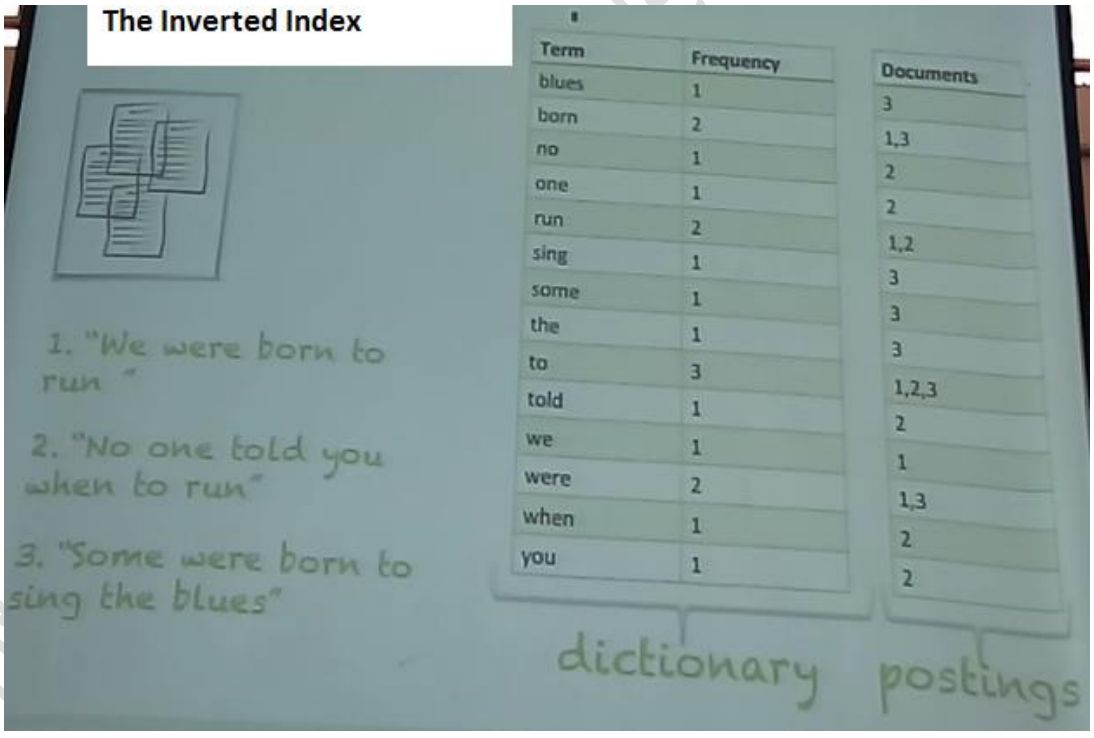
Note: Run this command on Master

ELK Software stack: Elastic Search, Logstash, Kibana

قبل از معرفی این استک ابتدا مختصری راجع به جستجو صحبت می کنیم. اساساً جستجو برمبنای ایندکس کردن می‌باشد. مثلا اگر یک سند داشته باشیم می‌توانیم برای آن یک term vector تعریف کنیم که مشخص کننده هر کلمه و فرکانس تکرار آن در سند می‌باشد. بعضی کلمات مثل am,is , are دارای فرکانس تکرار زیادی هستند ولی به تنهایی جستجو نمی شوند. بنابراین ایندکس فقط بصورت تک کلمه‌ای انجام نمی‌شود و گوگل تا ۵ کلمه را هم ایندکس می کند. حتی عناوین مقالات با رجوع بالا را نیز ایندکس می کند که می تواند تا ۱۵ کلمه باشد. این عمل در شکل زیر نشان داده شده است. اگر تعداد سند بیشتری داشته باشیم می‌توان برای هرکدام یک بردار مجزا در نظر گرفت. منتها در عمل جستجو آنچه در حقیقت استفاده می شود ایندکس معکوس (reverse index) می‌باشد. جدول مرتبط با ایندکس معکوس از روی جدول ایندکس ساخته می‌شود و تعیین می‌کند هر لغت در چه سند(هایی) وجود دارد.

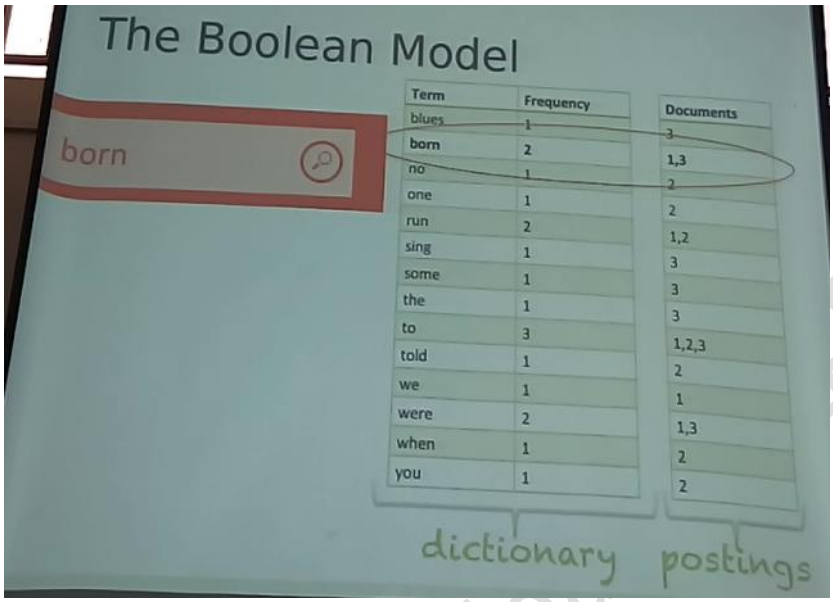


در شکل زیر نمونه یک عمل ایندکس معکوس نشان داده شده است. به فرایند تولید ایندکس معکوس، indexing می‌گویند یعنی عمل تولید تعداد زیادی inverted index. فرایند جستجو از روی ایندکس معکوس انجام می‌شود.

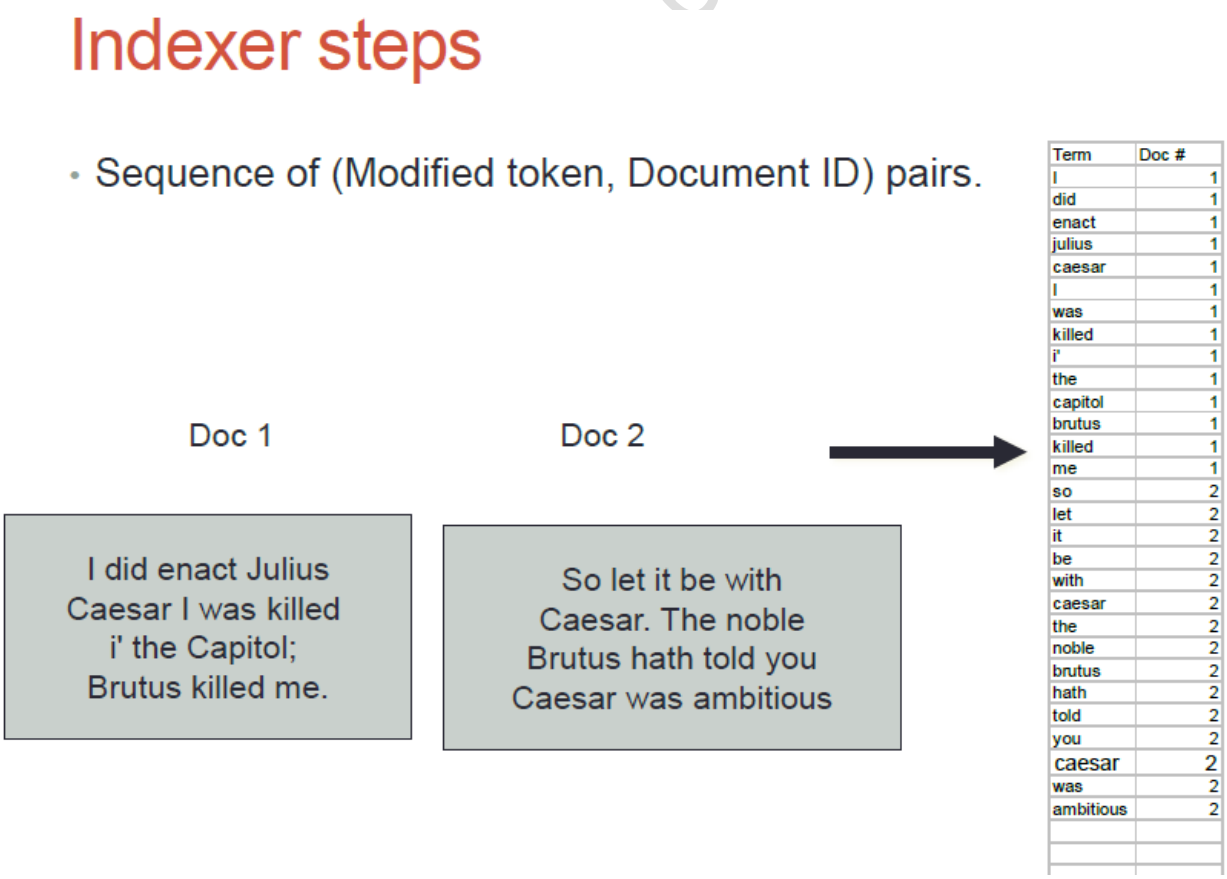


در عمل جستجو ممکن است از اعمال بولین استفاده شود. مثلا در سند کلمه we وجود دارد و کلمه were وجود دارد برای جستجو می توان اشتراک این دو را محاسبه کرد که هر دو در چه سندی وجود دارند اما ممکن هست هر دو در یک سند باشند ولی پشت سر هم نباشند بنابراین بهتر هست خود we were ایندکس شود که گفتیم گوگل تا ۵ کلمه را هم ایندکس می‌کند. در اینجا دو مفهوم precision, recall داریم که هر دو از جنس درصد هستند با این تفاوت که: precision یعنی از ۱۰۰ تا آیتم اولی که

گفتیم چند تای آنها درست است. Recall یعنی چند درصد از موارد درست را بازیابی کرده ایم. بین این دو می‌توان یک حدمیانه ای در نظر گرفت. یعنی لزوماً هر دو بالا نیستند و ممکن است precision بالا باشد ولی recall پایین باشد و برعکس.



در ۴ شکل زیر مراحل ایندکس کردن و ساخت reverse indexing برای دو سند doc1,doc2 نشان داده شده است.



- Sort by terms.

Core indexing step.

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

- Multiple term entries in a single document are merged.
- Frequency information is added.

Why frequency?
Will discuss later.

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

- The result is split into a *Dictionary* file and a *Postings* file.

Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

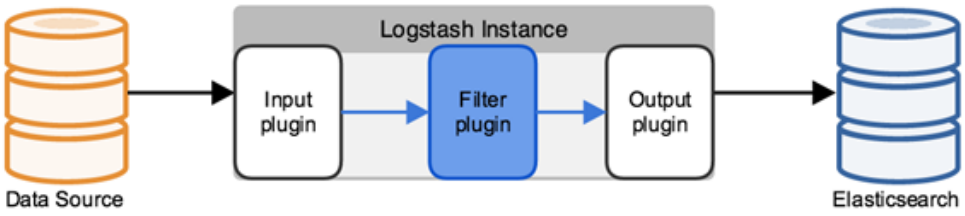
Elasticsearch ELK Software Stack:

Elasticksearch: کار ایندکس کردن و جستجو را انجام می‌دهد.

Logstash: ابزاری برای جمع‌آوری، تقویت، فیلتر کردن فوروارد داده‌ها مثل logها می‌باشد.

Kibana: ابزاری است که برای تصویری کردن داده‌ها و رسم نمودارها و مانیتورینگ استفاده می‌شود.

Logstash: یک نرم افزار open source برای جمع‌آوری، تبدیل، فیلتر و فوروارد داده‌ها از منابع ورودی به منابع خروجی می‌باشد. در JRUBY پیاده سازی شده است و روی JVM اجرا می‌شود. و دارای معماری بر پایه پیام ساده می‌باشد.



سرعت logstash بالاست و پیام‌ها بصورت استریم وارد پایپ logstash می‌شود و آنها را پردازش می‌کند. برای پیکربندی logstash باید از قالب فایل پیکربندی آن که بصورت زیر است استفاده کرد.

```
# This is a comment. You should use comments to describe
# parts of your configuration.
input {
  ...
}

filter {
  ...
}

output {
  ...
}
```

بخش ورودی input plugin تعیین کننده یک منبع خاصی است که توسط logstash خوانده می‌شود. منابعی را که می توان در این بخش تنظیم و پیکربندی کرد بسیار زیاد هستند که چند مورد آن بطور خلاصه در جدول زیر نشان داده شده اند. برای توضیح بیشتر به https://www.elastic.co/guide/en/logstash/5.6/input-plugins.html مراجعه کنید.

Plugin	Description	Github repository
beats	Receives events from the Elastic Beats framework	logstash-input-beats
cloudwatch	Pulls events from the Amazon Web Services CloudWatch API	logstash-input-cloudwatch
elasticsearch	Reads query results from an Elasticsearch cluster	logstash-input-elasticsearch
eventlog	Pulls events from the Windows Event Log	logstash-input-eventlog
exec	Captures the output of a shell command as an event	logstash-input-exec
file	Streams events from files	logstash-input-file
ganglia	Reads Ganglia packets over UDP	logstash-input-ganglia
gelf	Reads GELF-format messages from Graylog2 as events	logstash-input-gelf
kafka	Reads events from a Kafka topic	logstash-input-kafka
kinesis	Receives events through an AWS Kinesis stream	logstash-input-kinesis
pipe	Streams events from a long-running command pipe	logstash-input-pipe
redis	Reads events from a Redis instance	logstash-input-redis
relp	Receives RELP events over a TCP socket	logstash-input-relp
rss	Captures the output of command line tools as an event	logstash-input-rss
s3	Streams events from files in a S3 bucket	logstash-input-s3
stdin	Reads events from standard input	logstash-input-stdin
tcp	Reads events from a TCP socket	logstash-input-tcp
twitter	Reads events from the Twitter Streaming API	logstash-input-twitter
udp	Reads events over UDP	logstash-input-udp
unix	Reads events over a UNIX socket	logstash-input-unix

بخش فیلتر: یک filter plugin پردازش فوری روی یک رخداد را انجام می‌دهد. فیلترها هم بطور شرطی اعمال می‌شوند که وابسته به ویژگی‌های یک رخداد می‌باشد که چند مورد زیر را در جدول زیر نشان داه ایم. برای مشاهده موارد بیشتر به https://www.elastic.co/guide/en/logstash/5.6/filter-plugins.html مراجعه کنید.

Plugin	Description	Github repository
aggregate	Aggregates information from several events originating with a single task	logstash-filter-aggregate
alter	Performs general alterations to fields that the `mutate` filter does not handle	logstash-filter-alter
anonymize	Replaces field values with a consistent hash	logstash-filter-anonymize
cidr	Checks IP addresses against a list of network blocks	logstash-filter-cidr
cipher	Applies or removes a cipher to an event	logstash-filter-cipher
clone	Duplicates events	logstash-filter-clone
collate	Collates events by time or count	logstash-filter-collate
csv	Parses comma-separated value data into individual fields	logstash-filter-csv
date	Parses dates from fields to use as the Logstash timestamp for an event	logstash-filter-date
de_dot	Computationally expensive filter that removes dots from a field name	logstash-filter-de_dot

بخش خروجی: output plugin رخدادها را به مقصد خاصی می‌فرستد. خروجی ها مرحله نهایی خط لوله رخداد می‌باشند چند مورد از این خروجی‌ها در زیر نشان داده شده است برای مطالعه بیشتر به https://www.elastic.co/guide/en/logstash/5.6/output-plugins.html مراجعه کنید

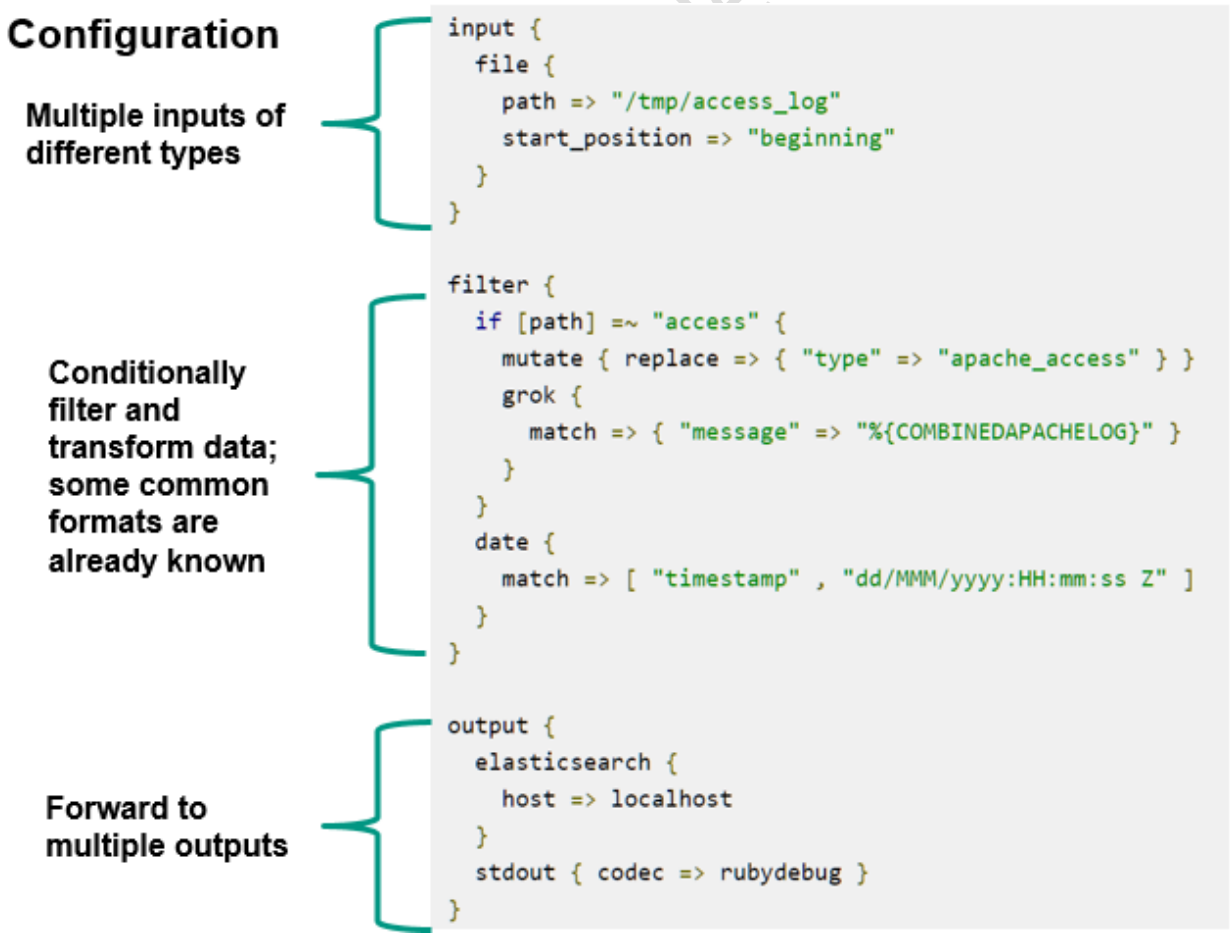
Plugin	Description	Github repository
boundary	Sends annotations to Boundary based on Logstash events	logstash-output-boundary
circonus	Sends annotations to Circonus based on Logstash events	logstash-output-circonus
cloudwatch	Aggregates and sends metric data to AWS CloudWatch	logstash-output-cloudwatch
csv	Writes events to disk in a delimited format	logstash-output-csv
datadog	Sends events to DataDogHQ based on Logstash events	logstash-output-datadog
datadog_metrics	Sends metrics to DataDogHQ based on Logstash events	logstash-output-datadog_metrics
elasticsearch	Stores logs in Elasticsearch	logstash-output-elasticsearch
email	Sends email to a specified address when output is received	logstash-output-email
exec	Runs a command for a matching event	logstash-output-exec
file	Writes events to files on disk	logstash-output-file

یک نمونه پیکر بندی logstash برای ورود داده ها از syslog بصورت زیر است:

Adding the 'syslog' input

```
input {
  beats {
    port => 5044
  }
  syslog {
    type => syslog
    port => 5514
  }
}
output {
  stdout { }
  elasticsearch { }
}
```

یک نمونه پیکربندی دیگر از logstash بصورت زیر است:



با اجرای این پیکربندی خروجی بصورت زیر خواهید داشت:

Console output processing apache log files

Run logstash with: bin/logstash -f logstash.conf

```
{
       "message" => "127.0.0.1 - - [11/Dec/2013:00:01:45 -0800] \"GET
       /xampp/status.php HTTP/1.1\" 200 3891 \"http://cadenza/xampp/navi.php\"
       \"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:25.0) Gecko/20100101
       Firefox/25.0\"",
    "@timestamp" => "2013-12-11T08:01:45.000Z",
      "@version" => "1",
          "host" => "cadenza",
      "clientip" => "127.0.0.1",
         "ident" => "-",
          "auth" => "-",
     "timestamp" => "11/Dec/2013:00:01:45 -0800",
          "verb" => "GET",
       "request" => "/xampp/status.php",
   "httpversion" => "1.1",
      "response" => "200",
         "bytes" => "3891",
      "referrer" => "\"http://cadenza/xampp/navi.php\"",
         "agent" => "\"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:25.0)
         Gecko/20100101 Firefox/25.0\""
}
```

نوع دیگری از پیکربندی می تواند بصورت زیر باشد:

Configuration for parsing syslog messages

Input filter receives messages directly from tcp and udp ports

Filter splits messages and adds fields

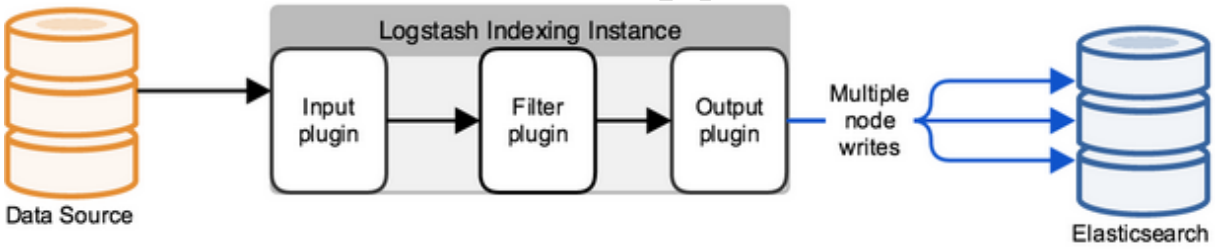
```
input {
  tcp {
    port => 5000
    type => syslog
  }
  udp {
    port => 5000
    type => syslog
  }
}

filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp} %{SYSLOGHOST:syslog_h
_program}(?:\[%{POSINT:syslog_pid}\])?: %{GREEDYDATA:syslog_message}" }
      add_field => [ "received_at", "%{@timestamp}" ]
      add_field => [ "received_from", "%{host}" ]
    }
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM  d HH:mm:ss", "MMM dd HH:mm:ss" ]
    }
  }
}

output {
  elasticsearch { host => localhost }
  stdout { codec => rubydebug }
}
```

Console output processing syslog messages

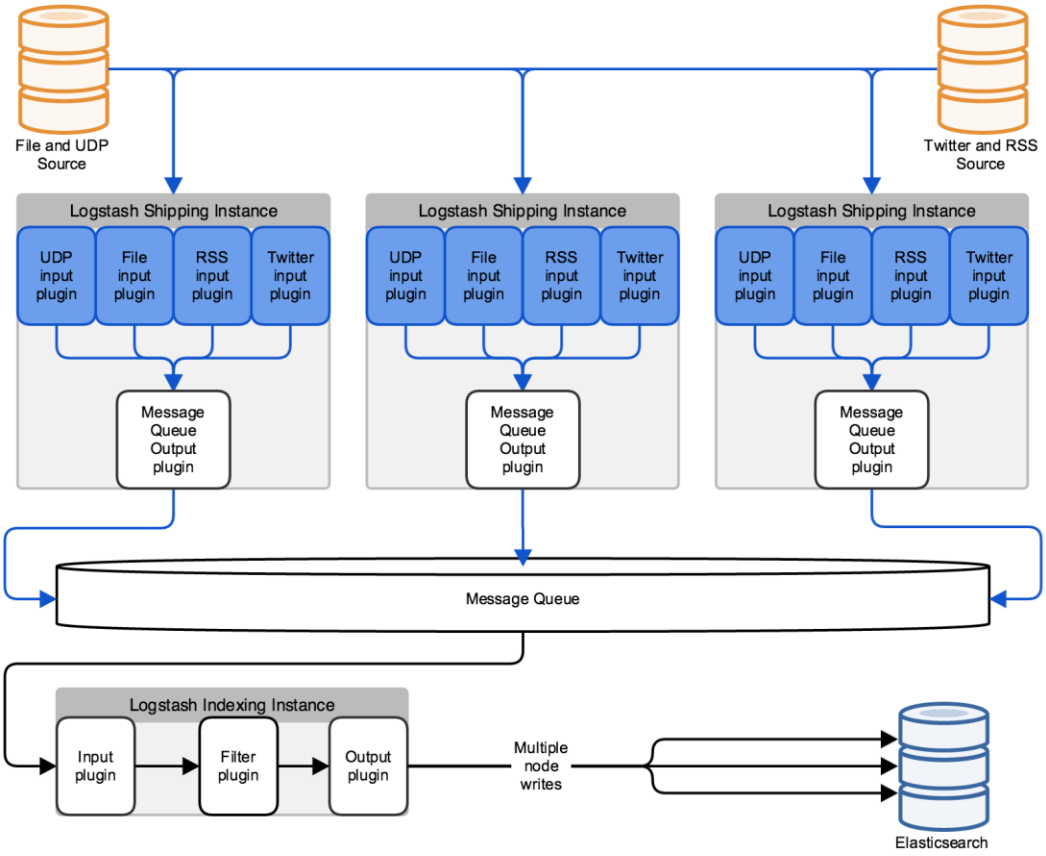
Run logstash with: bin/logstash -f logstash.conf

```
{
                 "message" => "Dec 23 14:30:01 louis CRON[619]: (www-data) CMD (php
                 /usr/share/cacti/site/poller.php >/dev/null
                 2>/var/log/cacti/poller-error.log)",
               "@timestamp" => "2013-12-23T22:30:01.000Z",
                 "@version" => "1",
                     "type" => "syslog",
                     "host" => "0:0:0:0:0:0:0:1:52617",
          "syslog_timestamp" => "Dec 23 14:30:01",
           "syslog_hostname" => "louis",
            "syslog_program" => "CRON",
                "syslog_pid" => "619",
            "syslog_message" => "(www-data) CMD (php /usr/share/cacti/site/poller.php
            >/dev/null 2>/var/log/cacti/poller-error.log)",
               "received_at" => "2013-12-23 22:49:22 UTC",
             "received_from" => "0:0:0:0:0:0:0:1:52617",
    "syslog_severity_code" => 5,
    "syslog_facility_code" => 1,
         "syslog_facility" => "user-level",
         "syslog_severity" => "notice"
}
```

می توان در logstash خروجی را در چند گره نوشت:



Logstash می‌تواند بصورت مقیاس‌پذیر با قابلیت دسترس بالا استفاده شود.

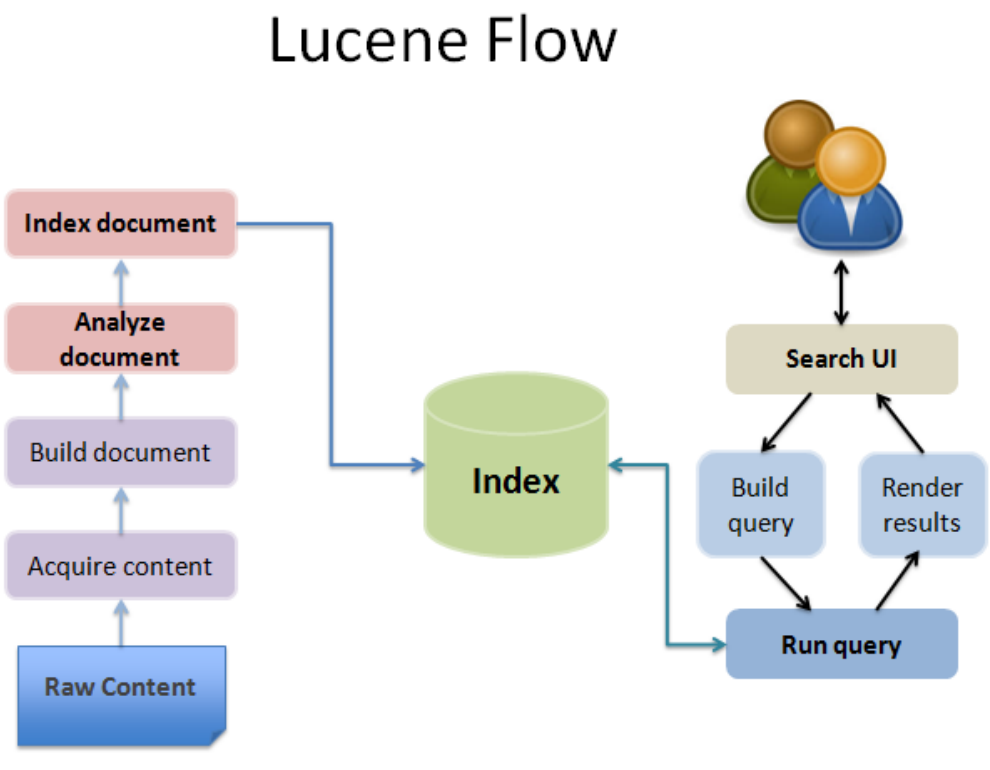


Elasticsearch

- Server environment for storing large scale structured index entries and query them
  - Written in Java
  - Based on Apache Lucene
  - Uses Lucene for index creation and management
  - Document-oriented (structured) index entries which can (but must not) be associated with a schema
  - Combines “full text”-oriented search options for text fields with more precise search options for other types of fields, like date + time fields, geolocation fields, etc.
  - Near real-time search and analysis capabilities
- Provides Restful API as JSON over HTTP

Scalability of Elasticsearch

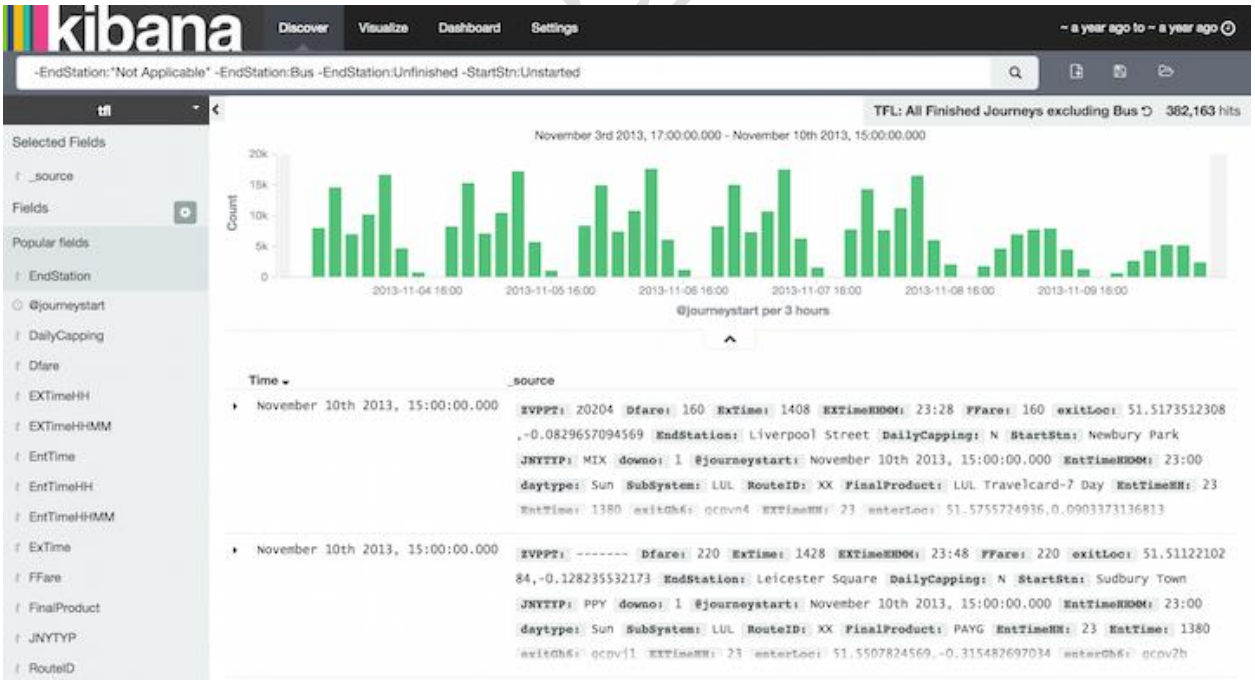
- Elasticsearch can run as one integrated application on multiple nodes of a cluster
- Indexes are stored in Lucene instances called “Shards” which can be distributed over several nodes
- There a two types of “Shards”
  - Primary Shards
  - Replica
- Replicas of “Primary Shards” provide
  - Failure tolerance and therefore protect data
  - Make queries (search faster) faster



Kibana: یک کاربرد است که به شما یک داشبورد می‌دهد و شما در آن مشخص می کنید از یک ایندکس یا دیتای elasticsearch ورودی بگیرد و آنها را بصورت نمودار و گراف به ما ارائه بدهد.

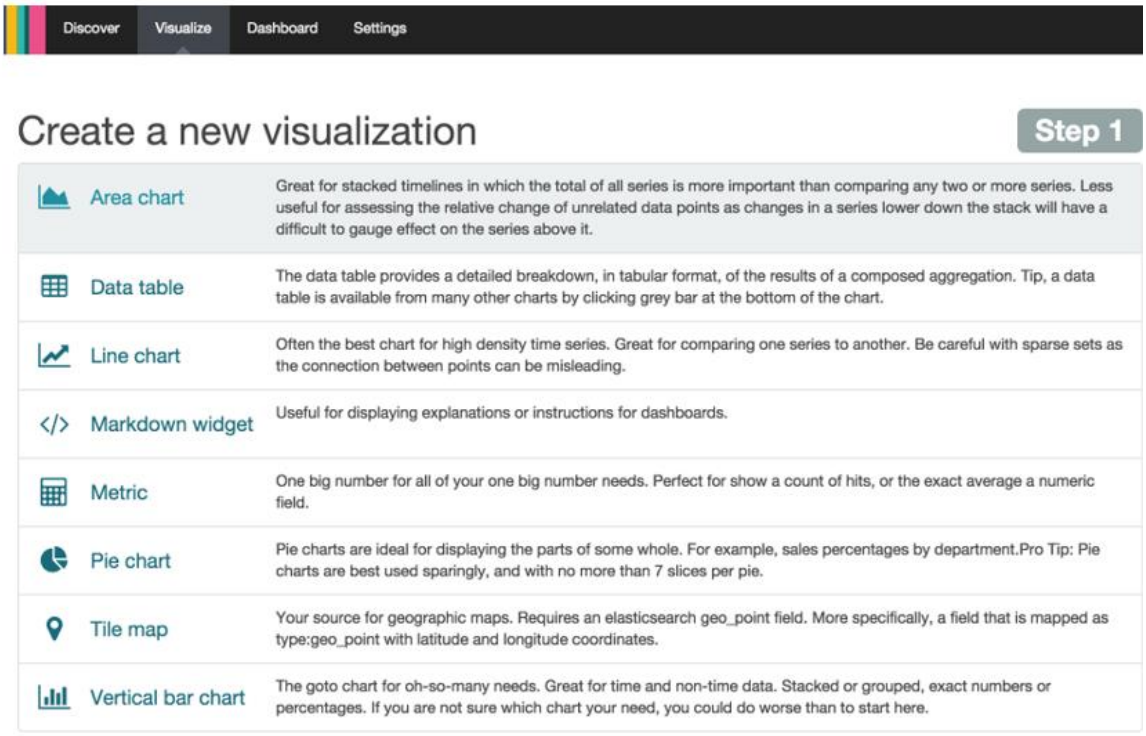
Kibana

- Web-based application for exploring and visualizing data
- Modern Browser-based interface (HTML5 + JavaScript)
- Ships with its own web server for easy setup
- Seamless integration with Elasticsearch



Create a visualization

Different types of visualizations



Some use cases of the ELK stack

- Log data management and analysis
- Monitor systems and / or applications and notify operators about critical events
- Collect and analyze other (mass) data
  - i.e. business data for business analytics
  - Energy management data or event data from smart grids
  - Environmental data
- Use the ELK stack for search driven access to mass data in web based information systems

Log data management and analysis

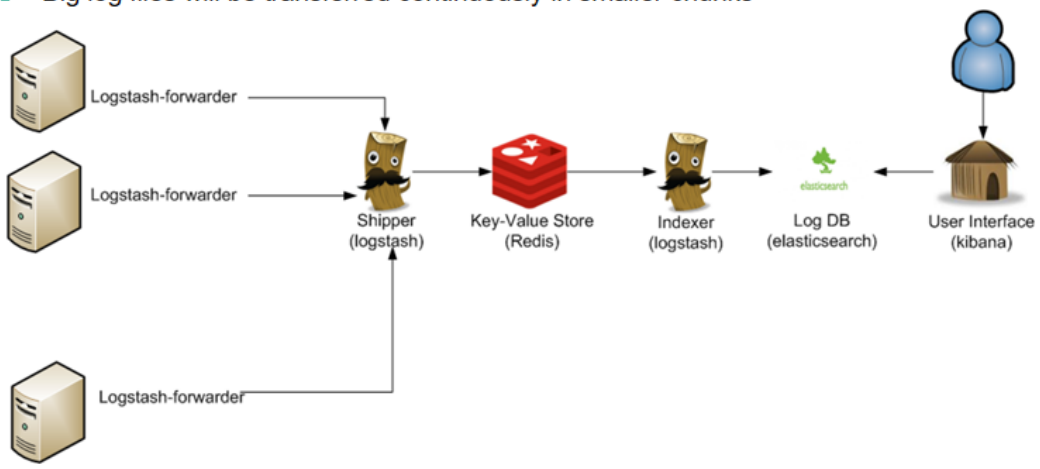
- Many different types of logs
  - Application logs
  - Operating system logs
  - Network traffic logs from routers, etc.
- Different goals for analysis
  - Detect errors at runtime or while testing applications
  - Find and analyze security threats
  - Aggregate statistical data / metrics

Problems of log data analysis

- No centralization
  - Log data could be everywhere
    - on different servers and different places within the same server
- Accessibility Problems
  - Logs can be difficult to find
  - Access to server / device is often difficult for analyst
  - High expertise for accessing logs on different platforms necessary
  - Logs can be big and therefore difficult to copy
  - SSH access and grep on logs doesn't scale or reach
- No Consistency
  - Structure of log entries is different for each app, system, or device
  - Specific knowledge is necessary for interpreting different log types
  - Variation in formats makes it challenging to search
    - Many different types of time formats

The ELK stack provides solutions

- Logstash allows to collect all log entries at a central place (e.g. Elasticsearch)
  - End users don't need to know where the log files are located
  - Big log files will be transferred continuously in smaller chunks



- Log file entries can be transformed into harmonized event objects
- Easy access for end users via Browser based interfaces (e.g. Kibana)
- Elasticsearch / Kibana provide advanced functionality for analyzing and visualizing the log data

Monitoring

- The ELK stack also provides good solutions for monitoring data and alerting users
  - Logstash can check conditions on log file entries and even aggregated metrics
  - And conditionally sent notification events to certain output plugins if monitoring criteria are met
    - E.g. forward notification event to email output plugin for notifying user (e.g. operators) about the condition
    - Forwarding notification event to a dedicated monitoring application
  - Elasticsearch in combination with Watcher (another product of Elastic)
    - Can instrument arbitrary Elasticsearch queries to produce alerts and notifications
    - These queries can be run at certain time intervals
    - When the watch condition happens, actions can be taken (sent an email or forwarding an event to another system)

مقایسه کلان داده با پردازش کلان[1]: همانطور که قبلا گفتیم پردازش‌های ما چهار نوع محدودیت دارند بعضی محدود به حافظه هستند، بعضی محدود به I/O، بعضی محدود به پردازش، بعضی محدود به ارتباطات[2]. در پردازش‌های کلان داده بیشتر نیازمند جمع آوری و جابجایی داده و دسترسی به داده‌ها، بارگذاری و کار با داده هستیم و کمتر درگیر پردازش داده هستیم در حالیکه برای مسائل HPC[3] بیشتر نیازمند پردازش سریع داده ها هستیم. این خود چگونگی نیاز ما به سخت افزار را تعیین می کند. همانطور که گفتیم برای مسائلی که نیازمند پردازش‌های خیلی سریع هستند بیشتر استفاده از سوپرکامپیوترها مناسب هستند و برای مسائل مرتبط با جابجایی داده‌ها بیشتر کامپیوترهای مقیاس ورهاوس یا دیتاسنترها مناسب هستند.

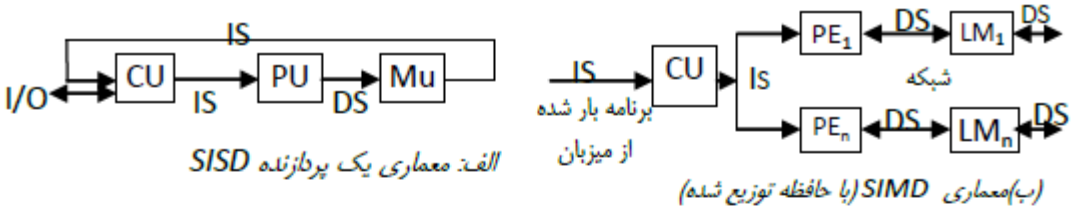
انواع مدل‌های محاسبات موازی: از لحاظ نوع موازات قابل دسترس در برنامه‌ها می‌توان آنها را به چند مدل تقسیم کرد:

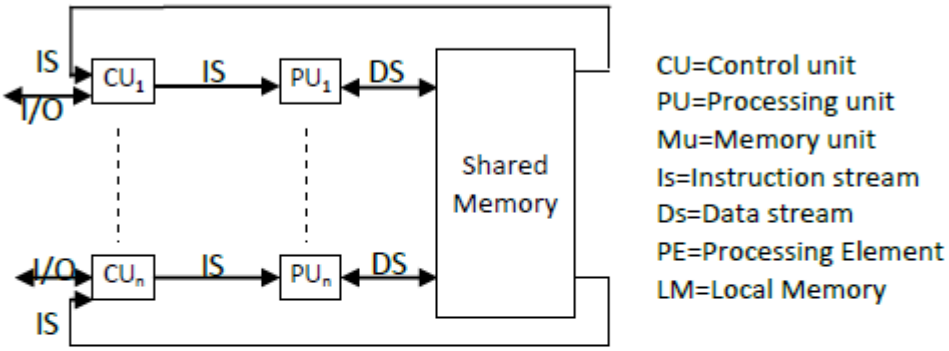
- موازات سطح داده: در این نوع موازات دستورات مشابهی بطور همزمان به چندین عنصر داده اعمال می‌شوند که به آنها SIMD[4] می‌گویند.
- موازات سطح کار (task): در این نوع موازات دستورات متفاوت بر روی داده‌های متفاوت اعمال می‌شوند که به آن MIMD[5] می‌گویند.
- یک برنامه چندین داده: در این نوع موازات، موازات روی یک سطح عمل محدود نشده است که به آن SPMD[6] می‌گویند. SPMD معادل MIMD می‌باشد چون هر برنامه MIMD می‌تواند یک SPMD تولید کند.

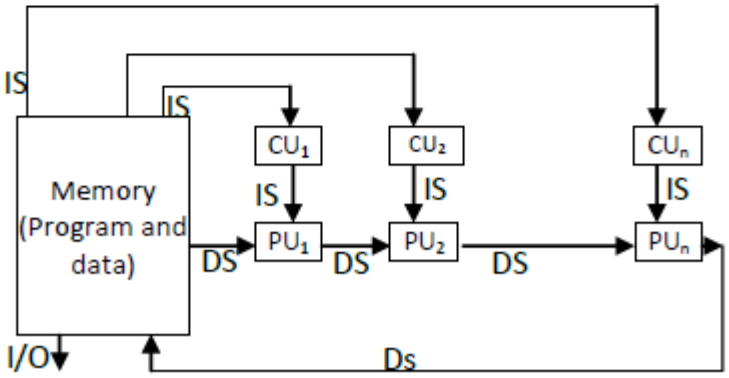
[1] Big Data Vs Big Process
[2] Memory bound, I/O bound, compute Bound, Communication Bound
[3] High Performance Computing
[4] Single Instruction Stream Multiple Data Stream
[5] Multiple instruction Stream Multiple Data Stream
[6] Single program multiple data.

- انتقال پیام[1]: برای موازات MIMD/SPMD می‌باشد.

از لحاظ سخت افزاری این طبقه بندی را Flynnبه صورت شکل زیر ارائه داد:







همانطور که در شکل الف مشاهده می‌شود سیستم SISD می باشد و یک رشته دستور روی یک رشته داده انجام می شود این نوع سیستم ها در CPU های اولیه استفاده می شد. در شکل ب نوع SIMD نشان داده شده است که سیستم همچنان تک هسته می‌باشد ولی دارای چندین واحد عملیاتی هست که می توانند بطور همزمان روی تعداد زیادی داده یک دستور را اجرا کنند. مثلا حلقه زیر را در نظر بگیرید:

```
For(i=0;i<1000;i++)
{C[i]=A[i]+B[i];}
```

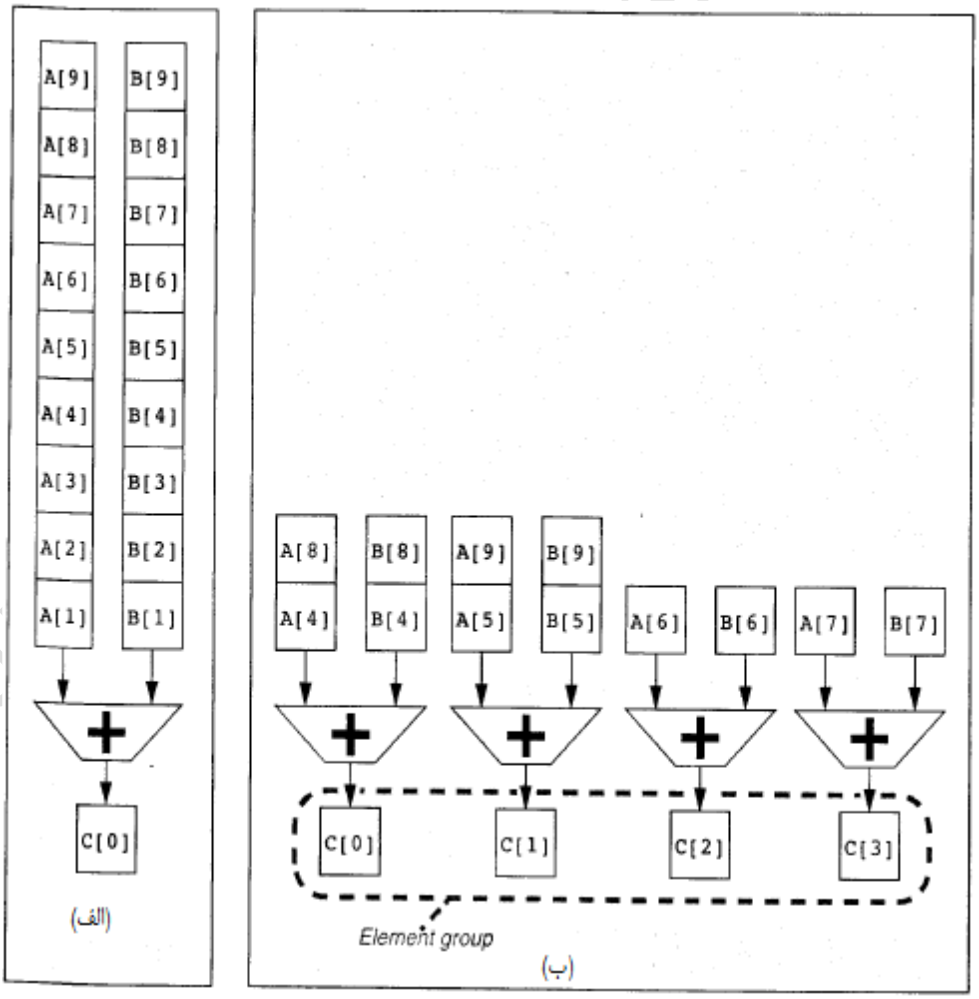
[1] Message Passing interconnection (MPI)

این حلقه یک دستور جمع را روی ۱۰۰۰ عنصر آرایه A,B انجام داده و در آرایه C قرار می دهد. اگر سیستم از نوع SISD باشد با ۱۰۰۰ بار اجرای این دستور نتیجه بدست می‌آید اما اگر در پردازنده ۴ واحد جمع کننده داشته باشیم حلقه را می‌توان بصورت زیر نوشت:

```
for(i=0;i<256;i++)
{
C[i*4]=A[i*4]+B[i*4];
C[i*4+1]=A[i*4+1]+B[i*4+1];
C[i*4+2]=A[i*4+2]+B[i*4+2];
C[i*4+3]=A[i*4+3]+B[i*4+3];
}
```

در هر بار اجرای حلقه چهار دستور جمع اجرا می‌شوند که هر دستور را یک واحد جمع انجام خواهد داد و این عمل بطور موازی انجام می‌شود و بدنه حلقه ۲۵۶ بار اجرا خواهد شد. این باعث افزایش سرعت تقریبا ۴ برابری در اجرای برنامه بصورت SIMD خواهد شد. نکته اینکه SIMD می تواند روی یک هسته پردازنده باشد. GPUها نوعی SIMD با تعداد واحدهای پردازشی بسیار زیاد هستند.

نمونه‌ای از اجرای یک حلقه را روی یک پردازنده SISD در مقایسه با SIMD با ۴ واحد جمع کننده در شکل زیر نشان داده‌ایم:

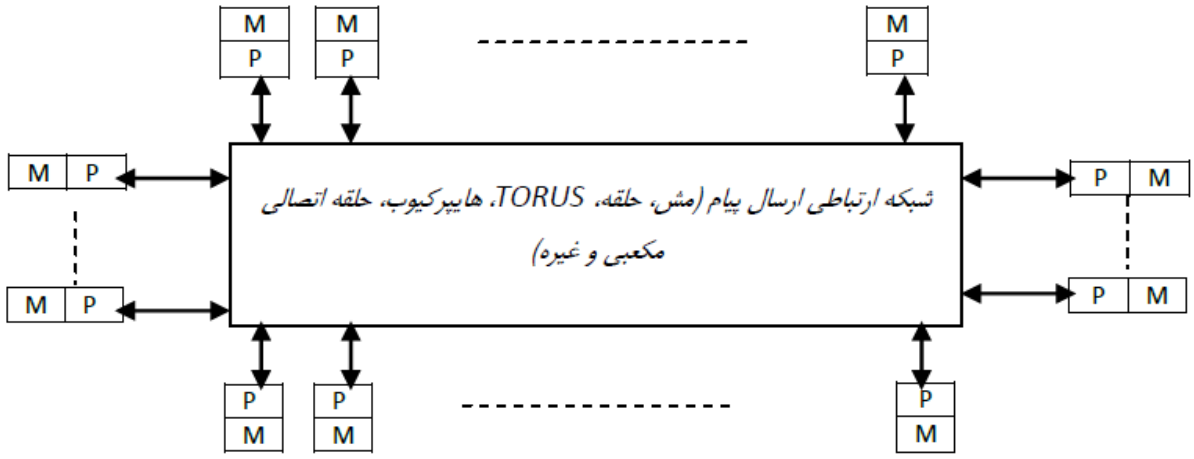


همانطور که در شکل ج مشاهده می‌کنید پردازش از نوع MIMD می باشد که امروزه بیشتر پردازنده‌ها از این نوع بوده و تعدادی هسته (core) در پردازنده وجود دارند و هر کدام می‌توانند روی یک thread کار کنند. در این حالت می توان برنامه را به چند thread تقسیم کرد که هر thread روی یک هسته بطور موازی با thread های دیگر اجرا شود. از مزایای دیگر MIMD موازات سطح کار هست که می توانند برنامه های مستقل به هر کدام از هسته ها برای اجرا داده شوند.

نوع MISD در عمل ساخته نشد. ولی بعضی ها خط لوله را نوعی MISD می‌دانند.

از لحاظ ارتباط بین هسته های پردازنده و حافظه دو نوع ارتباط داریم یکی از نوع حافظه مشترک است که در شکل فوق MIMD مشاهده می کنید. در سیستم های چندهسته حافظه اصلی بین همه هسته ها بطور مشترک استفاده می شود هرچند هر هسته ممکن است چند سطح حافظه کش خصوصی داشته باشد که معمولا در چندهسته ها مثل intel corei7 که ۴ هسته دارد کش L1 به هر پردازنده نزدیک با حجم کم و سرعت زیاد است و کش L2 نیز خصوصی هرپردازنده است ولی دارای حجم بیشتری نسبت به L1 ولی سرعت کمتر و انجمن پذیری بالاتر است. اما کش L3 بین هر چهار هسته مشترک است.

نوع دیگر ارتباط به انتقال یا ارسال پیام مشهور است که در چند پردازنده‌ها که پردازنده ها از هم جدا بوده به صورت شکل زیر استفاده می‌شود. در این شکل هر پردازنده حافظه اصلی خصوصی خودش (همچنین حافظه های کش خودش را دارد و ممکن است هر پردازنده چند هسته باشد) را دارد اما برای ارتباط بین پردازنده ها از سیستم‌های شبکه های میان ارتباطی یا گذرگاه مشترک و غیره استفاده می‌شود و تبادل اطلاعات با ارسال پیام انجام می‌شود.



چگونگی استفاده از موازات های مختلف در برنامه‌ها:

در حال حاضر سرعت Cpu ها به اندازه گذشته افزایش ندارد و سیستم‌های چند هسته رایج شده‌اند. برای استفاده از این قدرت مهم است که برنامه‌نویسان با دانش برنامه‌نویسی موازی آشنا باشند تا برنامه ای بنویسند که چند کار را بطور همزمان اجرا کند. در اینجا قصد داریم مقدمه‌ای مختصر راجع به OpenMp بیان کنیم که توسعه ای بر کامپایلرهای C/C++/Fortran است تا موازات را به کدهای موجود اضافه کند بدون اینکه نیاز به بازنویسی کل کد باشد. این توسعه در کامپایلرهای مختلف مثل GCC, Clang++, Solaris Studio, Intel C Compiler (icc), Microsoft Visual C++ since version 2005 پشتیبانی می‌شود.

مقدمه‌ای بر OpenMp در C++: OpenMp شامل مجموعه‌ای از compiler #pragmas می‌باشد که روش کار برنامه را کنترل می‌کند. Pragmas طوری طراحی شده‌اند که حتی اگر کامپایلر آن را پشتیبانی نکند، برنامه هنوز هم رفتار درست داشته باشد اما بدون موازات. در اینجا دو برنامه ساده را نشان می‌دهیم که از OpenMp استفاده می‌کند:

مثال: مقدار دهی به یک جدول بطور موازی (چند thread).

این کد مقداردهی جدول را به چند نخ تقسیم می‌کند، که بطور موازی اجرا می‌شوند. هر نخ (thread) بخشی از جدول را مقدار دهی می‌کند.

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

مثال: مقدار دهی یک جدول بطور موازی (یک نخ ,SIMD): این نسخه نیازمند پشتیبانی از OpenMp 4.0 استفاده از کتابخانه ممیز شناور موازی AMD ACML یا Intel SVML می‌باشد.

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp simd
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

مثال: مقدار دهی به یک جدول (چند نخ روی دستگاهی دیگر): OpenMP 4.0 از بارگذاری روی دستگاهی دیگر مثل GPU پشتیبانی می‌کند. بنابراین می‌تواند سه سطح موازات در یک برنامه موجود باشد: single thread processing multiple data، multiple threads running simultaneously، و multiple devices running same program simultaneously.

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp target teams distribute parallel for map(from:sinTable[0:256])
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

با توجه به مثال‌های فوق و نوع موازات می‌توان روش نوشتن کد موازی را بیان کرد:

ساخت بلوک موازی: یک بلوک موازی با نوشتن pragma ایجاد می‌شود. آن یک گروه از N نخ ایجاد می‌کند (که N در زمان اجرا تعیین می‌شود، که معمولاً مرتبط با تعداد هسته‌های CPU و عوامل وابسته دیگری می‌باشد)، که همه دستور بعدی با بلوک {} را اجرا می‌کنند. بعد از بلوک نخ‌ها به یک نخ تبدیل می‌شوند.

```
#pragma omp parallel
{
  // Code inside this region runs in parallel.
  printf("Hello!\n");
}
```

این کد یک مجموعه نخ ایجاد می‌کند که هر نخ یک کد مشابه را اجرا می‌کند. آنها hello چاپ می‌کنند و به خط بعدی می‌روند. برای یک سیستم دو هسته، متن دوبار تکرار می‌شود (ممکن است خروجی HeHllellolo نیز ایجاد شود که وابسته به سیستم است چون print بطور موازی اتفاق می‌افتند) بعد از } نخ ها به یک نخ join می‌شوند.

موازات شرطی با if: موازات را می‌توان با قرار دادن عبارت if در دستور موازات بصورت شرطی ایجاد کرد. مثل:

```
extern int parallelism_enabled;
#pragma omp parallel for if(parallelism_enabled)
for(int c=0; c<n; ++c)
  handle(c);
```

در این حالت اگر parallelism_enabled مقدار صفر برگرداند، تعداد نخ‌ها در گروه که حلقه را اجرا خواهند کرد همیشه یک خواهد بود.

Loop condtruct for: for یک حلقه for را می‌شکند بطوریکه هر نخ در گروه جاری یک بخش مختلف از حلقه را اجرا کند.

```
#pragma omp for
for(int n=0; n<10; ++n)
{
  printf(" %d", n);
}
printf(".\n");
```

این حلقه همیشه مقدار ۰ تا ۹ را چاپ می‌کند. اما ممکن است بطور تصادفی اجرا شود و خروجی به عنوان مثال 0 5 6 7 1 8 2 3 4 9 شود.

بطور داخلی، حلقه فوق بطور داخلی معادل با این می‌شود:

```
int this_thread = omp_get_thread_num(), num_threads = omp_get_num_threads();
int my_start = (this_thread  ) * 10 / num_threads;
int my_end   = (this_thread+1) * 10 / num_threads;
for(int n=my_start; n<my_end; ++n)
  printf(" %d", n);
```

بنابراین هر نخ یک بخش متفاوت حلقه را می‌گیرد، و بخش‌های خودشان را با موازت اجرا می‌کنند.

شما می‌توانید صریحا تعداد نخ‌هایی که باید در گروه تولید شوند را مشخص کنید با استفاده از num_threads:

```
#pragma omp parallel num_threads(3)
{
  // This code will be executed by three threads.

  // Chunks of this loop will be divided amongst
  // the (three) threads of the current team.
  #pragma omp for
  for(int n=0; n<10; ++n) printf(" %d", n);
}
```

زمانبندی (scheduling): الگوریتم زمانبندی برای حلقه for بصورت ضمنی کنترل می‌شود.

```
#pragma omp for schedule(static)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

پنج نوع زمانبندی وجود دارد: static, dynamic, guided, auto, runtime. و سه نوع scheduling modifier وجود دارد: monotonic, nonmonotonic, simd.

Static زمانبندی پیش فر ض است که در بالا نشان داده شد. بر اساس ورود به حلقه هر نخ بطور مستقل تصمیم می‌گیرد کدام بخش حلقه را آنها باید پردازش کنند. همچنین زمانبندی dynamic داریم:

```
#pragma omp for schedule(dynamic)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

در زمانبندی dynamic هیچ گونه ترتیب قابل پیش‌بینی در اختصاص عناصر حلقه به نخ های مختلف وجود ندارد. هر نخ از کتابخانه زمان اجرای OpenMP برای یک تکرار درخواست می‌کند، سپس آن را راه اندازی می کند و برای آینده نیز درخواست می‌دهد و غیره. این مفیدتر خواهد بود اگر با عبارت ordered ترکیب شود. یا وقتی که تکرارهای مختلف حلقه زمان های اجرای مختلفی بگیرند. همچنین می‌توانیم اندازه قطعه (chunk) را تعیین کنیم تا تعداد فراخوانی‌ها به کتابخانه را کاهش دهیم.

```
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

در این مثال، هر نخ ۳ اجرای حلقه را درخواست می‌کند. بطور داخلی حلقه فوق می‌تواند بطور داخلی معادل زیر باشد.

```
int a,b;
if(GOMP_loop_dynamic_start(0,10,1, 3, &a,&b))
{
  do {
    for(int n=a; n<b; ++n) printf(" %d", n);
  } while(GOMP_loop_dynamic_next(&a,&b));
}
```

عبارت Ordered: ترتیب اجرای تکرارهای حلقه نامشخص است و وابسته به شرایط زمان اجرا است. به هر حال می‌توانیم مجبور کنیم که ترتیب اجرای برنامه به ترتیب مشخصی باشد با استفاده از عبارت ordered:

```
#pragma omp for ordered schedule(dynamic)
for(int n=0; n<100; ++n)
{
  files[n].compress();

  #pragma omp ordered
  send(files[n]);
}
```

حلقه ۱۰۰ فایل را بطور موازی فشرده می‌کند، اما اطمینان می‌دهد که فایل‌ها به ترتیب سریال ارسال شوند.

اگر به یک نخ فایل شماره ۷ برای فشرده سازی اختصاص داده شده باشد، و فایل ۶ هنوز ارسال نشده باشد، نخ قبل از ارسال صبر خواهد کرد. عبارت ordered تضمین می‌دهد که فقط یک نخ وجود دارد که کارهای باشماره کمتر را تخصیص می‌دهد.

هر فایل فقط یک بار فشرده و ارسال می‌شود اما فشرده سازی بطور موازی انجام می‌شود. تر کیب های مختلف دیگری نیز برای این عبارات وجود دارد.

عبارت collapse: وقتی شما حلقه های تودرتو دارید، می‌توانید از عبارت collapse استفاده کنید تا نخ‌بندی را به چندین تکرار تودروتو اعمال کنید. مثلا:

```
#pragma omp parallel for collapse(2)
for(int y=0; y<25; ++y)
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
```

عبارت reduction: یک راهنمای خاص است که به کامپایلر دستور می‌دهد کدی تولید کند که مقادیر تکرارهای مختلف حلقه را با هم به ترتیبی خاص جمع (accumulate) کند.مثلا:

```
#pragma omp parallel for collapse(2)
for(int y=0; y<25; ++y)
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
```

Sections: گاهی می‌خواهیم مشخص کنیم که مثلا " این و این می توانند بطور موازی اجرا شوند". Sections برای تنظیم این مورد است مثلا:

```
#pragma omp sections
{
  { Work1(); }
  #pragma omp section
  { Work2();
    Work3(); }
  #pragma omp section
  { Work4(); }
}
```

این کد تعیین می‌کند که work1, work2+work3, work4 می‌توانند بطور موازی اجرا شوند اما work2 و work3 باید به ترتیب اجرا شوند. هر کدام از کارها فقط یکبار اجرا می‌شوند.

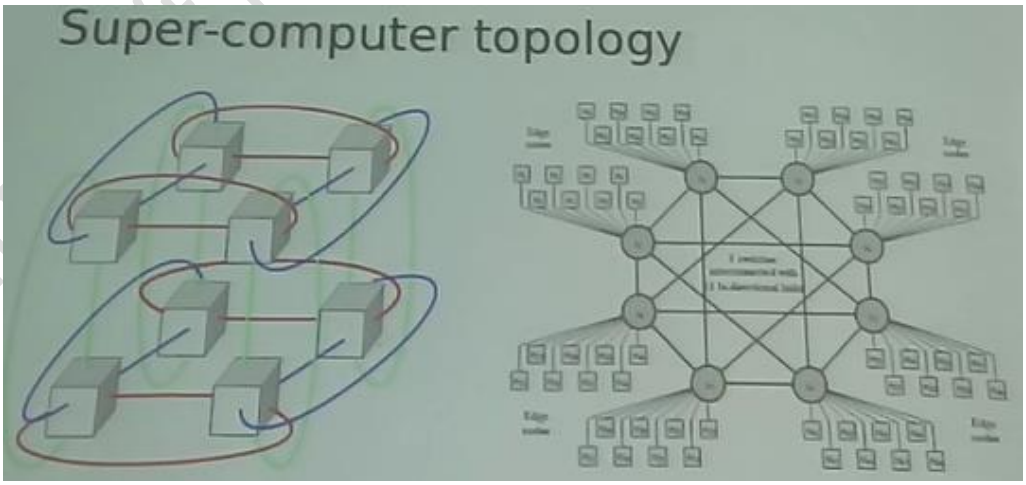
نکته: چرا با وجود این امکانات موازات بعضی موتورها مثل JVM از این موازات استفاده نمی‌کنند؟ زیرا در کلان داده بیشتر درگیر ذخیره و بارگذاری و کار با داده‌ها هستیم نه مسائل پردازش و افزایش سرعت پردازش.

پردازش‌های هم‌گرا در مقایسه با پردازش‌های واگرا[1]:

پردازش همگرا یعنی از هر نقطه‌ای شروع کنیم عاقبت به جواب می‌رسیم.

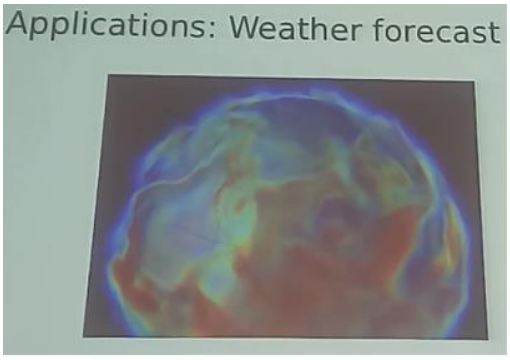
پردازش واگرا: اگر در اجرا یک خطا اتفاق بیافتد این خطا منتشر شده و بزرگ و بزرگتر خواهد شد.

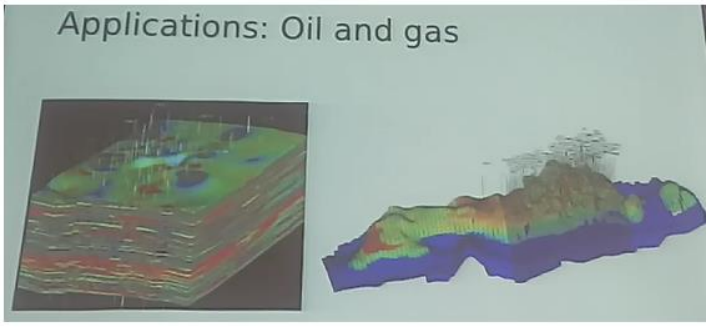
سوپرکامپیوترها در مقایسه با دیتاسنترها: قبلا توضیح داده شد که در سوپرکامپیوترها تعداد زیادی گره از نوع مشابه با درجه اتصال پذیری بالا به هم متصل شده‌اند و توپولوژی خاصی مثل تورس، فوق مکعب، شبکه چند سطحه، مش و ... بر اتصالات آنها حاکم است و بیشتر برای اهداف پردازشی با سرعت بالا طراحی شده اند. نمونه ای از توپولوژی‌های استفاده شده در سوپر کامپیوتر بصورت شکل زیر وجود دارد:

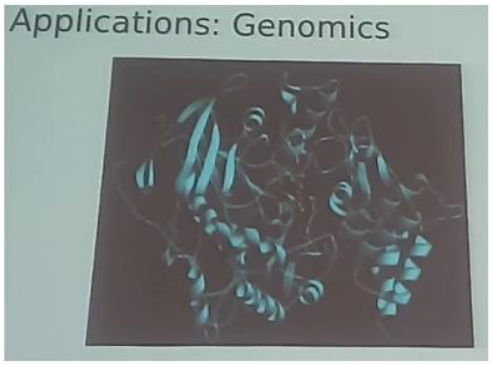


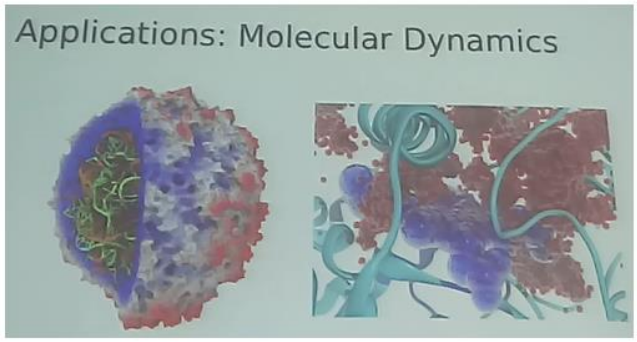
[1] Convergent Vs divergent computation

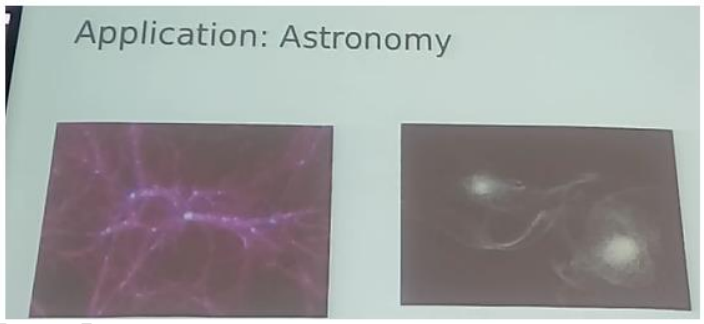
کاربرد و نوع استفاده از اینها وابسته به کاربرد و نیازهای ما می‌باشد بیشتر از دیتاسنترها برای ذخیره داده ها استفاده می شود و لزوماً سیستم ها مشابه نیستند. یا در ورهاوس ها سیستم‌های استفاده شده سیستم‌های معمولی هستند به تعداد زیاد. به عنوان مثال نمونه‌هایی از استفاده از سوپرکامپیوترها عبارتند از: پیش‌بینی آب و هوا، کاوش نفت و گاز، ژنتیک، حرکت‌های مولکولی، نجوم











Finding Out About the Environment

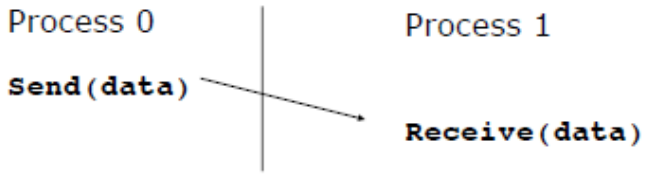
- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **MPI_Comm_size** reports the number of processes.
  - **MPI_Comm_rank** reports the *rank*, a number between 0 and size-1, identifying the calling process

Simple Program in C

```
#include "mpi.h"
int main(int argc, char *argv[])
{
  int rank, size, i, provided
  float A(10)
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE,
                  &provided);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (i=0; i<10; i++)
    A[i] = i * rank;
  printf("My rank %d of %d\n", rank, size );
  printf("Here are my values for A\n");
  for (i=0; i<10; i++) printf("%f ", A[i]);
  printf("\n");
  MPI_Finalize();
}
```

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called `MPI_COMM_WORLD`.

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive.
- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by (**start**, **count**, **datatype**).
- The target process is specified by **dest**, which is the rank of the target process in the communicator specified by **comm**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (on source and tag) message is received from the system, and the buffer can be used.
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE.
- status contains further information
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.

Send-Receive Summary

- Send to matching Receive

MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

- Datatype
  - Basic for heterogeneity
  - Derived for non-contiguous
- Contexts
  - Message safety for libraries
- Buffering
  - Robustness and correctness

Retrieving Further Information

- `Status` is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Adding Communication

- Test yourself here. Take our original program and change it to do the following:
- Process 0 (i.e., the process with rank 0 from MPI_Comm_rank) sets the elements of A[i] to i, using a loop.
- Process 0 sends A to all other processes, one process at a time, using MPI_Send. The other processes receive A, using MPI_Recv.
  - The MPI datatype for "float" is MPI_FLOAT
  - You can ignore the status return in an MPI_Recv with MPI_STATUS_IGNORE
- The program prints rank, size, and the values of A on each process

One Answer to the Question in C (part 1)

```
#include "mpi.h"
int main(int argc, char *argv[])
{
  int rank, size, i, provided
  float A(10)
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE,
                  &provided);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

One Answer to the Question in C (part 2)

```
if (rank == 0) {
   for (i=0; i<10; i++)
     A[i] = i;
   for (i=1, i<size; i++)
       MPI_Send(A, 10, MPI_FLOAT, i, 0,
                MPI_COMM_WORLD);
} else {
   MPI_Recv(A, 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
}
printf("My rank %d of %d\n", rank, size );
printf("Here are my values for A\n");
for (i=0; i<10; i++) printf("%f ", A[i]);
printf("\n");
MPI_Finalize();
}
```

Timing MPI Programs

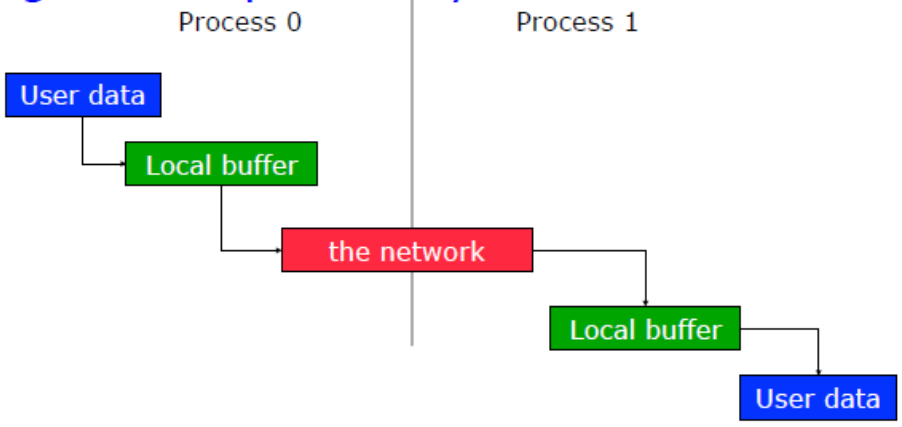
- The elapsed (wall-clock) time between two points in an MPI program can be computed using **MPI_Wtime**:

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "time is %d\n", t2 - t1 );
```

- The value returned by a single call to **MPI_Wtime** has little value.
- The resolution of the timer is returned by **MPI_Wtick**
- Times in general are local, but an implementation might offer synchronized times.
  - For advanced users: see the MPI attribute MPI_WTIME_IS_GLOBAL.

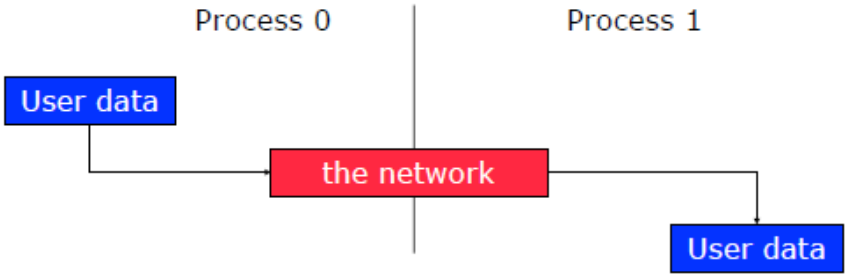
Buffers

- When you send data, where does it go? One possibility is:



Avoiding Buffering

- It is better to avoid copies:



This requires that `MPI_Send` wait on delivery, or that `MPI_Recv` return before transfer is complete, and we wait later.

Blocking and Non-blocking Communication

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called “unsafe” because it depends on the availability of system buffers

Solutions to the "safety" Problem

- Order the operations more carefully
- Supply receive buffer at same time as send (`MPI_Sendrecv`)
- Supply own buffer space (`MPI_Bsend`)
- Use non-blocking operations
  - Safe, but
  - not necessarily asynchronous
  - not necessarily concurrent
  - not necessarily faster

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on.

```
MPI_Request request;
MPI_Isend(start, count, datatype,
    dest, tag, comm, &request);
MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request);
MPI_Wait(&request, &status);
```

- One can also test without waiting:

```
MPI_Test(&request, &flag, &status);
```

ضمیمه ۱: جلسه آقای ستیانی

بیت کوین زیردرختی از بلاک چین هست. در آینده هر کس بیشتر داده داشته باشد پولدار هست. در ابتدا وب غیرمتمرکز بود سپس متمرکز شد و مجددا توزیع شده است. دیتا ها به third party فروخته می شود. هر کاربر فیس بوک ۱۲۰ دلار هست. ایران ۲ دلار، آمریکا بالاترین قیمت. در متمرکز یک سرور بود و یک central point داشتیم. گسترش سخت و یک SPOF داریم. فیس بوک حدود ۲۰ عدد دیتاسنتر دارد.توزیع شده در decentralize است. اگر یک نود خراب شود بقیه خراب نمیشوند. مثل بیت کوین مثل ایتریوم یا Steemit. در steemit در مقابل محتوا پول میگیرید.

مقایسه وب ۲ و وب ۳:

وب ۲: یک آمازون EC2 میگرفتیم یا ایران پارس آن لاین.فایل ها را قرار می دادیم. پول درمی اوریم

وب ۳: Ethtereum, Truebit. ذخیره سازی در IPFS/Flecoin. داده های خارجی اوراکل. وب ۳ بصورت نظیر به نظیر peer-to-peer می‌باشد. با شرکت خاصی قرار داد نبستیم. و با یکسری اجتماعات براساس منافع قرارداد می بندیم. Opensource است. هر فرد هم سرمایه گذار هست هم مشتری. بنابراین از کپی کاری به خاطر قدرت اجتماع جلوگیری می‌شود. برای جابجایی بیت کوین جماعت آن را تایید می‌کنند. برای هک کردن باید قدرتی بیشتری از ۵۱٪ افراد اجتماع داشته باشیم بنابراین کلاهبرداری ضعیف می‌شود. Decentralized hash table (DHT). تراکنش بصورت secure اتفاق می‌افتد.

مهمترین ویژگی No central point of failure می‌باشد. در این حالت دیگه کسی نمی‌تواند سیستم را down کند. در این حالت بجای درخواست به یک سرور در IPFS می‌توانیم هر بخش از اطلاعات را از جاهای مختلف بگیریم. به content دسترسی داریم نه درخواست http. IPFS توزیع شده است. مثلا یک عکس در کل سیستم های یوزرها ذخیره می‌شود. در IPFS آدرس دهی براساس محتوا می‌باشد. حملات DDoS غیرممکن به نظر می‌رسد. در حالیکه HTTP متمرکز است، که آدرس دهی براساس مکان است و مستعد حملات DDoS است.

IPFS	HTTP
Websites/apps have no centralized origin server	Based on a centralized origin server
Utilizes content-addressing	Location based addressing
DDoS attacks seem impossible	Susceptible to DDoS attacks

IPO=initial public offering، با هم مشترک می‌شویم و با چند نفر مستخدم شروع می کردیم و کار را به سرمایه گذار های خطر پذیر ارائه می‌کردیم. و آن را گسترش می‌دهیم.

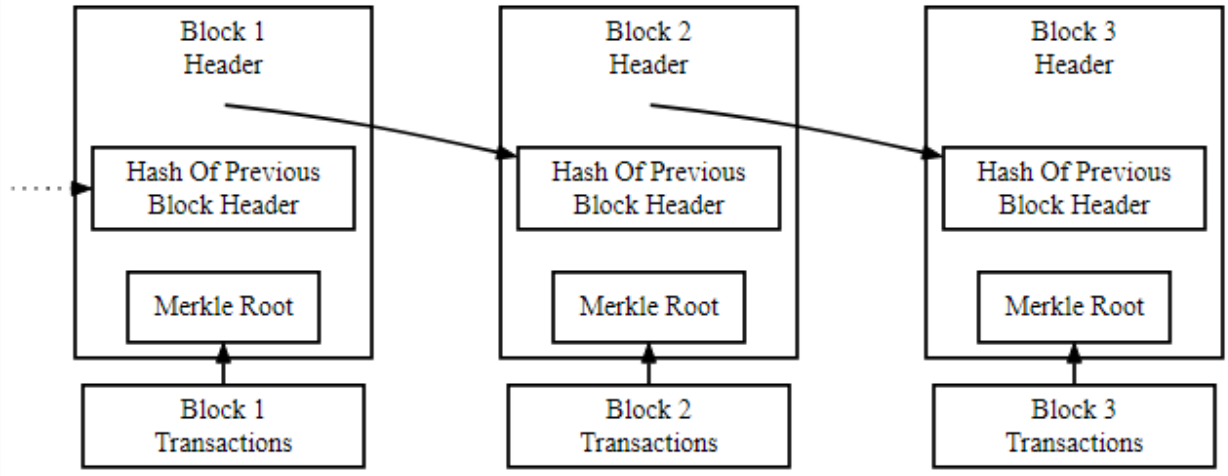
ICO=Initial coin offering

در این روش افراد به وسع خودشان سرمایه گذاری می کنند از طریق بیت کوین، از طریق اتریوم و ...

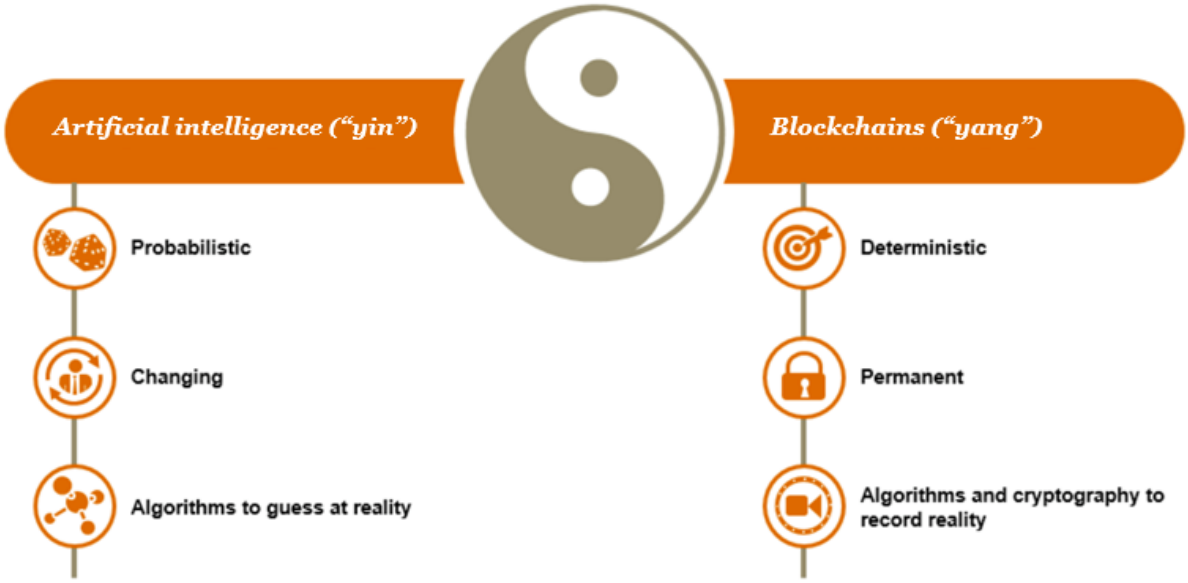
در Web3 کل تیم باید مهندس باشند و Whitepaper می‌نویسیم و به اجتماع گزارش می‌دهیم مزایای آن را مطرح می کنیم و با بیت کوین سرمایه گذاری می‌کنیم. در بیت کوین تورم نداریم به بانک اعتماد نمی کنیم بلکه به اجتماع اعتماد می‌کنیم.

وقتی یک هکر بلاک چین را هک کرد کل شبکه تصمیم گرفتند کل بلاک چین را برگردانند. هم هکر دستگیر شد و هم پول برگشت کرد.

Deep learning +AI+Blockchain:



Simplified Bitcoin Block Chain



The Yin & Yang

- AI is about machines doing educated guesswork for us
- Machines "learn" by processing training datasets
- Probabilistic methods that head in the direction of most likely reality
- AI algorithms are central to goals

- Blockchains are more about validation, permanence and greater degrees of certainty and control. .
- An immutable, verifiable Ledger
- Intelligence discovers opportunities & blockchains validate them and
- Validates identities & transactions, automate the validation and generate the audit trail that goes back as far as the chain does.
- Blockchain algorithms merely a means of permanently validating and recording transaction truth.
- blockchain data = canonical reference for who did what when, why, & for how much.

منابع و مراجع:

۱- مطالب ارائه شده در کلاس کلان داده

2- Big Data Related Technologies, Challenges and Future Prospects, Min Chen, Shiwen Mao, Yin Zhang,Victor C.M. Leung, SPRINGER BRIEFS IN COMPUTER SCIENCE, 2014

3- Understanding big data, Analytics for Enterprise Class Hadoop and Streaming Data, Chris Eaton, Drik Deroos, Tom Deutsch, George Lapis, Paul Zikopoulus, McGraw Hill, 2012

4- Bigtable: A Distributed Storage System for Structured Data, Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, ffay,jeff,sanjay,wilsonh,kerr,m3b,tushar,_kes,gruberg@google.com, *Google, Inc.2006*

۵- *کتاب معماری کامپیوتر پیشرفته- گردآوری و تالیف دکتر محمود فتحی-رضا سعیدی نیا*

6- https://www.youtube.com/watch?v=hMt9yFp0JKM

7- https://www.youtube.com/watch?v=B_HTdrTgGNs

8- Cassandra - A Decentralized Structured Storage System, Avinash Lakshman, Facebook, Prashant Malik, Facebook

9- Scala programming Language, www.tutorialspoint.com

10- https://spark.apache.org/docs/latest/streaming-programming-guide.html

11- https://data-flair.training/blogs/install-apache-spark-multi-node-cluster

12- The Logstash Book, James Turnbull, November 26, 2016

13- https://github.com/logstash-plugins/logstash-input-example/

14- https://www.elastic.co/guide/en/logstash/current/config-examples.html

15- https://www.elastic.co/guide/en/logstash/5.0/deploying-and-scaling.html

16- https://bisqwit.iki.fi/story/howto/openmp/