Multi-Level Elasticity for Data Stream Processing

IEEE Transactions on Parallel and Distributed Systems

2019

18 citations

France

در محیط‌های دیتااستریمی[1] هدف پردازش حجم زیادی از داده‌ها با تاخیر پایین و منابع محدود می‌باشد. با استفاده از Apache Storm یک استراتژی مدیریت کشسان[2] ارائه می‌دهیم که درجه موازی اجزای برنامه‌ها را تعدیل می‌کند در حالی که به صراحت به سلسله مراتب کانتینرهای اجرایی (ماشین‌های مجازی، پراسس و نخ‌ها) می‌پردازد. نشان می‌دهیم که فراهم‌سازی کانیتنر اشتباه منجر به پایین آمدن کارایی می‌شود و راهکاری ارائه می‌دهیم که کانتینر با کمترین هزینه (مصرف کمینه منابع) برای افزایش کارایی فراهم می‌آورد. معیارهای نظارت[3] خود را توصیف می‌کنیم و نشان می‌دهیم که چگونه ویژگی های یک محیط اجرا را در نظر می‌گیریم. همچنین یک ارزیابی تجربی با اپلیکیشن‌های دنیای واقعی ارائه می‌دهیم که کاربرد رویکرد ما را تأیید می‌کند.

همانطور که می‌دانیم بیگ دیتا در حال افزایش است. دو راهکار برای پردازش بیگ دیتا موجود می‌باشد:

- Batch processing: دیتا را در ابتدا ذخیره کرده و سپس با استفاده از مدل‌های برنامه‌پذیر مقیاس‌پذیر مثل Google's MapReduce آن را پردازش می‌کنیم. اما مشکلاتی مانند هزینه ذخیره سازی و نیاز به تولید اطلاعات و استفاده از این اطلاعات در مرحله‌های بعد می‌باشد. به همین دلیل روش دوم ارائه شد.
- Stream processing : در اینجا دیگر داده ها با الگوی خاصی نمی‌آیند و ممکن است در برخی لحظات ترافیک بالا و در لحظات دیگر ترافیک پایین باشد. لذا لازم است منابع پردازشی مان را مطابق با وضعیت های موجود افزایش یا کاهش دهیم که به این روش Elastic گویند و دیگر static کاربرد نخواهد داشت.

کانتینرهای اجرایی متفاوت هزینه متفاوت و عملکرد متفاوتی خواهند داشت. فراهم‌کردن یک کانتینر با ظرفیت بالا که معمولاً ماشین مجازی می‌باشد از نظر زمان و منابع هزینه بیشتری از یک کانتینر با ظرفیت پایین مثل نخ یا پراسس دارد. هدف ما فراهم‌کردن منابع با کمترین هزینه به منظور ارضای خواسته‌های اپلیکیشن می‌باشد. در یک سلسه مراتب از کانتینرها، راه حل مامنابع کانتینرهای با ظرفیت بالاتر را با استفاده از ترکیب کانتینر با ظرفیت کمتر استفاده می کند.

نوآوری‌هایی که ما ارائه می‌دهیم:

- ارایه استاندارد برای تشخیص ازدحام محلی در اپلیکیشن دیتااستریمینگ: ایده این است که یک کامپوننت اپلیکیشن به bottleneck می‌رود اگر نتواند تمامی داده‌های ورودی را جذب کند.
- فراهم آوری کشسان منبع چندسطحی: ارائه یک استراتژی کشسان که برای کانتینرهای اجرایی چندین سطح در نظر می‌گیرد. ایده این است که حداقل تعداد کانتینرهای سنگین (مانند ماشین های مجازی یا فرآیندها) را فراهم کنیم و با تکثیر تعداد کانتینرهای

[1] dataStream
[2] elastic
[3] Monitoring metrics

سبک وزن (رشته ها) استفاده از منابع را به حداکثر برسانیم. مهمتر از آن، تاثیر کانتینرهای مختلف اجرایی بر روی عملکرد را نیز در نظر می‌گیریم، زیرا استفاده از منبع نادرست می‌تواند عملکرد را کاهش دهد. ما یک استراتژی ساده و در عین حال کارآمد برای محک زدن برنامه‌ها[4] و کشف تجربی پیکربندی کمینه(ارزان‌ترین) برای یک محیط معین طراحی می‌کنیم.

- ادغام شفاف[5] در ApacheStorm: با استفاده از رابطهای برنامه‌نویسی آن می‌توان کانتینرهای اجرایی را در سطح‌های مختلف فراهم آورد. به صورت خودکار اپلیکیشن‌ها را محک می‌زنیم. کمترین کانفیگ لازم برای پشتیبانی یک محیط‌کار را ارائه می‌دهیم و کنترلرهای کشسان برای مدیریت bottleneck های اپلیکیشن ارائه می‌دهیم. و درنهایت در دومورد واقعی اپلیکیشن مان را مورد امتحان قرار می‌دهیم. اولین مورد یک شناسایی حمله منع خدمت توزیع‌شده می‌باشد. دومین مورد نیز تحلیل آنلاین نتایج تولید شده توسط برنامه شبیه‌ساز COMD می‌باشد.

سیستم و مدل‌ما:

مدل ما خصوصیات متداول محیطهای پردازش استریم موجود را ضبط می‌کند.مدل ما اپلیکیشن را به صورت یک گراف غیرحلقوی جهت‌دار نشان می‌دهد که نودهای آن عملگرهای محاسباتی و یالها دیتااستریم می‌باشند. عملگرهای واسبته به اپلیکیشن هستند و میتوانند فیلترهای ساده تا تبدیلات داده پیچیده باشند. استریم‌های داده از تاپلهای دودویی کلید و پیلود تشکیل شده‌اند که کلید برای نگاشت و گروه بندی می‌باشد و پیلود هم اون اطلاعاتی است که بایستی پردازش کنیم. ورودی سیستم داده‌های تولیدی توسط سیستم‌های خارجی می‌باشد.خروجی نیز می‌تواند یک نمودار بصری یا یک سری اطلاعات باشد که برروی دیسک ذخیره می‌شوند. مدل ما به صورت یک کلاستر می‌باشد که شامل چندین نود اجرایی و یک نود مدیریت مستر می‌باشد که مانیتور نودهای دیگر و مقیاس و بازیابی را مدیریت می‌کند. نودها یک سلسله مراتبی از کانتینرهای اجرایی را برای اجرای برنامه‌ها فراهم می‌آورند. به طور معمول مثلاً یک نود می‌تواند یک ماشین مجازی باشد که چندین پردازه چند نخی را اجرا می‌کند. محاسبات برنامه نیز می‌توانند توسط چندین نمونه[6] به صورت موازی اجرا شوند. این نمونه‌ها به کوچکترین کانتینرها(همان پایین ترین کانتینرها در سلسله مراتب کانتینرها) نگاشت شده اند. تغییر تعداد نمونه ها مستلزم تغییر در سطح موازی بودن کانتینرهای اجرایی است که می تواند در ریزدانگی[7]های مختلف انجام شود. به‌عنوان مثال اگر نمونه‌ها توسط نخ‌ها اجرا شوند، ممکن است یک نح جدید در یک پردازه موجود، در یک پردازه جدید تر بر روی همان ماشین یا یک ماشین جدید اجراشود.

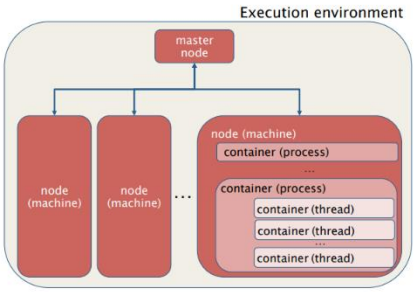

Fig. 2: Stream processing execution environment

معیارهای کارایی: معیار اصلی ما در دیتااستریم‌ها ظرفیت اپلیکیشن برای پردازش ورودی و تولید یک نتیجه در زمان مناسب می‌باشد. در گراف توپولوژی اولیه بلکه مشخص کردن فعالیت پردازش هر نمونه جدا می‌باشد. ما از معیارهای نمونه برداری زیر استفاده می‌کنیم:

- تعداد تاپل‌های دریافتی برای یک نمونه C در یک دوره تناوبی T. ($received_{T(c)}$)
- تعداد تاپل‌های پردازش شده برای یک نمونه C در یک دوره تناوبی T. ($processed_{(c)}$)
- فعالیت پردازشی[8]: سلامت یک نمونه را با استفاده از معیارهای قبلی محاسبه می‌کند:

[4] Benchmark
[5] Transparent Integration
[6] instance
[7] granularity
[8] Processing Activity

$$health\mathrm{T}(\mathrm{c}) = \frac{\mathrm{processedT}(\mathrm{c})}{\mathrm{recievedT}(\mathrm{c})}$$

اگر برای یک نمونه سلامتی آن برابر ۱۰۰ درصد باشد، می‌تواند تمامی تاپل‌های ورودی را پردازش کند. *اگر پایین‌تر بود، تاپل‌های منتظر برای پردازش می‌باشند.* و اگر همین وضعیت ادامه داشته‌باشد، تبدیل به گلوگاه کارایی برنامه می‌شود.

این که یک برنامه عملکرد خوبی داشته باشد و محدودیت‌های زمانی را بتواند ارضا کند، بستگی به منابع موجود هر نود، ارتباطات بین شبکه ای و تنوعاتی که در ترافیک و محاسبات برنامه دارد. در این مقاله ما اپلیکیشن استریم را به این صورت در نظر می‌گیریم که حجم زیادی از پیغام‌های سایز کوچک را پردازش می‌کند (به طور مثال پست‌های توییتر، پیغام‌های سنسورها و...). کارایی این برنامه‌ها محدود به منابع محاسباتی می‌باشد و از اشباع شبکه صرفنظر می‌کنیم. منابع محاسباتی هم شامل این موارد زیر می‌باشند:

- درگیری پردازنده: $\mathrm{totalCPU_{T(c)}}$ میانگین میزان درگیری پردازنده برای هر نود اجرایی در یک دوره زمانی T.
- درگیری حافظه: $\mathrm{totalMem_{T(c)}}$ میانگین میزان حافظه استفاده شده برای هر نود اجرایی در یک دوره زمانی T.

اگر هیچ کدام از این دو مورد از یک مقدار آستانه‌ای بیشتر نبود، به این معناست که نود منابع استفاده نشده دارد و می‌تواند به کانتینرهای اجرایی دیگر اختصاص دهد.اگر یکی از این دو مورد اشباع شده بود نیاز به یک تصمیم مقیاس مناسب دارد.

Apache Storm : یک فریمورک پردازش دیتااستریم تحمل خطاپذیر توزیع‌شده. به اپلیکیشن‌های بر پایه آن ***topologies*** گفته می‌شود که از ***spout*** ها که منابع دیتااستریم هستند و ***bolt*** ها که دیتااستریم‌ها را پردازش می‌کنند(دریافت، تبدیل و انتقال). از چهار سطح اجرایی مختلف تشکیل شده‌است.

- ***Nodes*** : ماشین‌های مجازی یا فیزیکی که اپلیکیشن بر روی آنها اجرا می‌شود.
- ***Workers***: پردازه‌ها مثلاً ماشین‌های مجازی جاوا که تعداد آنها برای هر نود توسط ادمین مشخص می‌شود.
- ***Executers***: ***worker*** ها شامل ***executer*** ها می‌باشند مثلاً نخ‌های جاوا: که ***task*** های که مطابق نمونه‌های ***spout*** ها و ***bolt*** ها هستند را اجرا می‌کنند.
- ***Tasks*** اون چیزایی که واقعن اجرا می‌شون و یک ***task*** توسط یگ ***executor*** اجرا می‌شود. تعداد ***task*** ها همون تعداد ***exector*** هایی است که این توپولوژی می‌تواند داشته باشد. لذا تعداد ***task*** به منظور عدم محدودسازی کارایی، باید بالا باشد.

از ***storm*** که بر روی یک کلاستر مجازی شده توسط ***openstack*** می‌باشد، استفاده می‌کنیم. در ذیل یک اپلیکیشن فیلتر را در نظرگرفته‌ایم که شامل یک ***spout*** می‌باشد که رشته‌هایی را به سمت یک ***bolt*** ارسال می‌کند و آنها را با یکسری الگوهای از پیش تعریف شده تطبیق می‌دهد. ***Spout*** را جوری تنظیم کرده‌ایم که ترافیک را با یک نرخ خطی افزایشی ارسال می‌کند.***bolt*** نیز بر روی یک ***node*** پیاده کرده‌ایم و تعداد ***task***،***executer***،***worker*** ها را افزایش می‌دهیم. نتایج به شرح زیر می‌باشد:

افزایش موازی سازی وظایف بر روی عملکرد برنامه هیچ تاثیری نخواهد داشت. در واقع ***tasks***، ماهیت متفاوتی در مقایسه با ***worker*** ها و executor ها دارند.Task ها بخش‌های منطقی محاسبات هستند و یک task واحد توسط یک ***exectuter*** واحد اجرا می‌شود.

موازی‌سازی ***executor*** تا زمانی که تعداد ***executer*** ها از تعداد هسته‌های پردازنده(منابع محدود) تجاوز نکرده باشد، باعث افزایش کارایی bolt خواهد شد

موازی‌سازی worker: فراهم نمودن بیش از یک ***worker***باعث کاهش کارایی خواهد شد. توضیح در این واقعیت نهفته است که چندین worker حافظه بیشتری را اشغال می کنند و سریعتر به محدودیت گره می رسند.

از این آزمایش‌ها نتیجه می‌گیریم که موازی‌سازی کانتینرهای مختلف storm نتایج متفاوتی را روی کارایی پردازش خواهد گذاشت.

استراتژی *elastic* چندسطحی: همانند شکل رسم شده در بالا، اپلیکیشن را فرض می‌کنیم که روی کلاستری از نودها در حال اجرا می‌باشد. محیطمان *multi-tenant* نمی‌باشد و نودها اختصاری اپلیکیشن می‌باشند. اپلیکیشن روی سلسله مراتبی از کانتینرها پیاده شده است و هر نمونه توسط یک کانتینر انحصاری پشتیبانی می‌شود. زمانی که به کانتینرهای اجرایی منابع محاسباتی کافی داده نشده‌باشد، امکان وقوع ازدحام وجود دارد.

Scaleout: اگر اپلیکیشن ورودی‌ها را با سرعت بالایی پردازش نکند، نیاز به تخصیص منابع بیشتری خواهد داشت

Scalein: اگر اپلیکیشن بتواند با منابع کمتری داده‌ها را پردازش کند.

بارهای پردازشی دیتااستریمی، میان نمونه‌ها تقسیم می‌شود.

مدام فعالیت محاسباتی یک اپلیکیشن را با استفاده از معیارهای معرفی‌شده در صفحه ۳ مانیتور می‌کند. و منابع را اختصاص می‌دهد. ما یک کنترلر *elastical* می‌سازیم که هم محیط اجرایی هدف و هم اپلیکیشن هدف را در نظر داشته باشد. روشی که ارائه می‌دهیم شامل سه گام می‌باشد و موارد زیر را به ترتیب در نظر می‌گیرد:

- تاثیر کارایی کانتینرهای اجرایی مختلف برای فراهم آوری منابع موردنیاز اپلیکیشن
- کشف کانفیگ‌های اپلیکیشن مناسب درخور *workload* با *benchmark* خودکار مداوم اپلیکیشن: با یک کانفیگی شروع می‌کنیم که درآن همه عملگرها موازی نشده‌اند و روی یک ماشین اجرا می‌شوند. هدف این است که با تغییر سطح *workload* ورودی‌ها، به کشف میزان موازی سازی و کانتینرهای متناظر برای عملگرهای اپلیکیشن بپردازیم.
طریقه کار *benchmark*: یک تزریق کننده داریم که نرخی از تاپل‌ها را در ثانیه به سمت اپلیکیشن ارسال می‌کند. یک ثابت بار اولیه و یک ثابت بار ماکزیمم در نظر می‌گیریم. بار اپلیکیشن را در یک دوره با استفاده از یک گام افزایش بار افزایش می‌دهیم و اپلیکیشن را برای تصادم در هر دوره مانیتور میکنیم(برای هر کدام از اپراتور های موازی جدا). ترتیبی که برای عملگرها در نظر گرفته می‌شود، بر اساس فاصله‌شان تا *source* های اپلیکیشن می‌باشد(به این دلیل که کارایی نمونه‌هایی که پایین دست‌تر میباشند را تحت تاثیر قرار میدهد).
یک نمونه وقتی دچار مشکل می‌شود که نرخ تاپل‌های پردازش شده آن از نرخ دریافتی‌ها کمتر باشد. وقتی که از یک مقدار آستانه‌ای کمتر شد، می گوییم نمونه دچار تصادم شده است و سطح موازی بودن اپراتورهای متناظرش را افزایش می‌دهیم.
عملیات افزایش تخصیص منابع به دنبال نودهای با کمترین سطح بار و میزان منابع کافی می‌گردد. اگر یک نود میزان مصرف پردازشگر یا حافظه مصرفی آن از حدی بالاتر رفت، معلوم می‌شود که توان اجرای یک نمونه عملگر عملیاتی جدید را نخواهد داشت. لذا یک ماشین مجازی جدید و یک سلسه مراتب کانتینری جدید فراهم آورده می‌شود. اگر نود منابع داشت، نمونه جدید با درنظر گرفتن کم هزینه‌ترین کانتینر به منظور افزایش کارایی پیاده می‌شود.
- ساخت و تولید یک کنترلر *elastic* مخصوص برنامه: با دانستن سطح *workload* های ورودی که به طور موفقیت آمیزی با کانفیگ‌های مختلف پردازش شده‌اند، می‌توان آستانه *workload* را تعیین کرد.

نتیجه‌گیری:

در این مقاله تاثیر کانتینرهای اجرایی مختلف را برای یک سیستم پردازش استریم کشسان بررسی کردیم. به صورت صریح سلسه مراتب کانتینرهای اجرایی(ماشین‌ها، پردازه‌ها و نخ‌ها) را در نظر گرفتیم و نشان دادیم که هزینه فراهم نمودن آنها با هم تفاوت خواهد داشت. علاوه بر آن نشان دادیم که فراهم نمودن کانتینری با نوع اشتباه، می‌تواند عملکرد اپلیکیشن را پایین بیاورد. یک مدیریت کشسان منابع برای *Apache Storm* ارائه دادیم که یک اپلیکیشن را با ارزان ترین کانفیگ منبع توسعه می‌دهد (از نظر مقیاسی بزرگ می‌کند). پیاده سازی ما برای اپلیکیشن ها *transparent* می‌باشد از آنجایی که از *API* ها *ApacheStorm*استفاده می‌کند