Distribuera mera - Spark och Hadoop utan Big Data

En undersökning i hur verktyg designade för Big Data kan användas med små datamängder

LUNDS UNIVERSITET
Campus Helsingborg

LTH Ingenjörshögskolan vid Campus Helsingborg

Institutionen för datavetenskap

Examensarbete:
Oscar Nihlgård

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Lunds universitet
Lund 2016

Sammanfattning

Distribuerad parallellisering innebär att en uppgift parallelliseras på flera datorer, till exempel exekvering av kod eller lagring av data. Konceptet går hand i hand med big data – extrema datamängder som inte kan bearbetas av en ensam dator. De mest etablerade verktygen för distribuerad parallellisering är därav verktyg avsedda för big data.

Denna rapport undersöker huruvida två sådana verktyg, Spark (distribuerad exekvering) och Hadoop Distributed File System (distribuerad lagring), är lämpade att använda även för små datamängder. Distribuering kan potentiellt vara ett billigt och skalbart sätt arbeta även vid mindre datamängder.

Undersökningen görs primärt genom prestandatester. Som ett sidospår implementeras ett abstraktionslager som möjliggör att kod kan köras både distribuerat och lokalt på effektivt sätt, genom att Javaströmmar används som en lokal motsvarighet till Spark. Detta innebär att för små arbetsuppgifter som enbart ibland är lämpade för distribuering ibland så går det att vid kodens körning avgöra vad som är lämpligast och välja därefter.

Rapportens slutsats är att dessa verktyg mycket riktigt kan användas för små mängder data, och även då exekveringstiden för en lokal implementation är mycket kort (under en minut).

Nyckelord: big data, hadoop, spark, hdfs, distribuering

Abstract

Distribution as a concept means that a task (for example, data storage or code execution) is parallelized on multiple computers. It goes hand in hand with the concept of big data – extreme amounts of data that can't be processed by a single computer. Because of this, the most established tools for distributed parallelization is tools that are designed to handle big data.

This thesis explores whether two such tools, Spark (distributed code execution) and Hadoop Distributed File System (distributed data storage), are also suited for handling smaller amounts of data. Distribution is a potentially cheap and scalable way of working even for small amounts of data.

The primary method of the report is performance tests. As a side track, an abstraction layer that allows for code to be executed either distributed or locally is implemented by using Java streams as a local equivalent of Spark. With this abstraction layer small tasks that are only sometimes suited for distribution can choose the best alternative at run time.

It is concluded that these tools can be useful even for small amounts of data, and even when the execution time for a non-distributed solution is very short (under a minute).

Keywords: big data, hadoop, spark, hdfs, distributed computing

Förord

Denna rapport har skrivits i samarbete med företaget Cybercom som har bidragit med handledning, arbetsutrymme och hårdvara.

Särskilt tack till

Jens Nilsson

för praktisk hjälp vid bland annat prestandatesterna

Elin A. Topp & Mathias Haage

som varit handledare respektive examinator för projektet

Ivar Grimstad & Dan Nilsson på Cybercom

för handledning och för att de gjort denna rapport möjlig

Innehållsförteckning

1. Inledning

Big data är ett begrepp som blir allt vanligare, men vad som egentligen avses är ofta otydligt. Det rör sig så klart om datamängder av extrema storlekar, men det är egentligen inte storleken i sig som är avgörande för om ett dataset kan klassas som big data. Ur ett strikt teknisk perspektiv innebär big data en enda sak: data som på grund av sin storlek bör bearbetas med distribuerade metoder.

Distribuering innebär att ett kluster av datorer (noder) tillsammans uppfyller ett syfte som traditionellt hade uppfyllts av en enda dator. Allmänt kan det sägas finnas två typer av distribuering: lagring och exekvering. Distribuerad lagring innebär att data lagras över flera datorer och distribuerad exekvering innebär att kod exekveras parallellt på flera datorer.

Distribuerad exekvering är konceptuellt likt lokal parallellisering på processorkärnor, men med några viktiga skillnader. Vid processorparallellisering är det nödvändigt att varje tråd använder så lite delad data som möjlig, då delad data måste användas på ett atomärt sätt (det vill säga flera trådar kan inte arbeta med samma data samtidigt). Vid distribuering finns samma problem, men kostnaden för att garantera atomära operationerna på den delade datan är mycket högre då kommunikationen mellan noderna sker över ett nätverk.

För att uppnå effektiv distribuerad exekvering räcker det därav inte med att begränsa den delade datan mellan noderna, den måste elimineras helt. Detta skapar en begränsning på hur distribuerbara algoritmer kan utformas. Förutsättningarna för distribuerad och lokal parallellisering är alltså olika.

1.1 Syfte med rapporten

De verktyg som finns för distribuering har vanligtvis ett fokus på att arbeta med stora datamängder, så kallad big data. Distribuering kan dock vara ett billigt och effektivt sätt att arbeta med även små datamängder på ett skalbart sätt.

Denna rapports syfte är att att undersöka när distribueringsverktygen Spark och Hadoop Distributed File System är lämpligt att använda för “små situationer”. Med små situationer avses situationer med liten datamängd och kort exekveringstid.

Två olika metoder används för att uppnå detta syfte. Dels görs ett antal prestandatest för att se hur väl Spark presterar i dessa små situationer. Dels utvecklas ett abstraktionslager som tillåter att kod som arbetar mot det kan köras både lokalt och distribuerat.

Abstraktionslagret är relevant då ett hinder vid distribuering i små situationer är att distribuering bara är meningsfullt *ibland* (det kan till exempel vara så att datamängden eller exekveringstiden varierar, eller att antalet tillgängliga noder varierar). Att skriva kod som kan köras både distribuerat och lokalt är därmed potentiellt en förutsättning för att Spark skall vara användbart för dessa situationer.

2. Bakgrund

Det finns två typer av distribuering: lagring och exekvering. I rapporten kommer två olika verktyg för dessa ändamål användas. För lagring kommer filsystemet Hadoop Distributed File System (HDFS) att användas och för exekvering kommer verktyget Spark att användas. HDFS valdes då det är det mest etablerade distribuerade filsystemet för big data, samt att det fungerar bra tillsammans med Spark.

Båda dessa verktyg är specifikt avsedda för att hantera så kallad big data. Distribuering på grund av big data har egentligen bara en viktig skillnad gentemot distribuering i allmänhet: feltolerans. När big data bearbetas så kan det ske med tusentals noder parallellt, vilket leder till att även om risken för hårdvarufel är låg för en given nod så är risken mycket hög för hela klustret. Att kunna hantera hårdvarufel på ett bra sätt är därför centralt för ett big data verktyg [3].

2.1 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) [17] är ett distribuerat filsystem baserat på Google File System (GFS) [8]. Det tillhör ett ekosystem av verktyg vid namn Hadoop som länge varit starkt dominerande inom big data området. Filsystemet går inte att köra självständigt, utan det arbetar ovanpå något annat godtyckligt filsystem [3].

Distribuerad läsning är relevant även för relativt små datamängder. Då en ensam hårddisk är begränsad till drygt 100MiB/s (så klart högre om ett SSD-minne används, men det har en begränsad relevans för stora datamängder) kan själva inläsningen av data bli en flaskhals även för relativt små datamängder. I bästa fall kan distribuerad läsning kombineras med distribuerad exekvering, så att varje nod bearbetar den data som är lagrad lokalt (så att ingen data behöver skickas mellan noderna).

Ett minimalt HDFS-kluster består av en namnnod och ett antal datanoder. Namnnoden hanterar all metadata i systemet, medans datanoderna lagrar den faktiska datan. Det är bara namnnoden som har kännedom om hur filsystemet faktiskt ser ut, datanoderna har bara rå fildata.

Att distribuera skrivning är något som endast är relevant för de mest extrema situationerna. Även om en hårddisk skriver långsammare än den läser så är det mycket få saker som genererar data så snabbt att det blir en flaskhals. Det är likväl något som *kan* distribueras, vilket också har gjorts av till exempel Facebook i Messenger [4].

2.1.1 Block

En hårddisk lagrar data i form av block, vilket är den minsta mängden data som kan skrivas eller läsas åt gången. Även ett filsystem arbetar med block, vars storlek är en multipel av hårddiskens blockstorlek. Hårdiskblocken är vanligen 512 bytes, medan filsystemblocken vanligen är några få kilobyte [3] (4KiB är till exempel den rekommenderade blockstorleken för NTFS [7]).

Blockstorleken är relevant vid läsning av filer. För varje block måste hårddisken söka upp hårddiskblockets fysiska position. Det betyder att tiden för inläsning av data består av två delar; dels sökandet efter blocket och dels den faktiska läsningen. Filsystemblock är implementerade så att de består av hårddiskblock som ligger i sekvens. Detta innebär att vid läsning av ett filsystemsblock krävs bara en sökning, även om det är flera hårddiskblock som läses.

Om en liten fil skall läsas in är det lönsamt med en låg blockstorlek - det leder till att mindre data måste läsas på grund av att ingen "onödig" data läses (då filsystemet alltså måste läsa in hela block) [3].

Det är även så att många filsystem är designade med antagandet att de flesta filer inte använder så många filsystemblock. Till exempel kan filsystemen i ext-familjen nämnas där söktiden ökar linjärt med antalet block så länge antalet block är mindre än tretton. När antalet block är högre än så börjar söktiden öka snabbare [9].

Även HDFS använder sig av block, men de är avsevärt större än i traditionella filsystem: 128MiB som standard. Det är viktigt att förstå att blockstorleken i HDFS inte påverkar läshastigheten på riktigt samma sätt som blockstorleken i lokala filsystem. På grund av att datan i HDFS måste lagras i ett underliggande filsystem så är det fortfarande blockstorleken i det underliggande filsystemet som är avgörande för den lokala läshastigheten.

Den stora blockstorleken i HDFS får alltså andra effekter än vad den hade fått i ett lokalt filsystem. I beskrivningen av GFS anges bland annat dessa två skäl för den stora blockstorleken [8]:

- Minska kommunikationen med namnnoden - desto mer data som kan bearbetas som en enhet desto mindre kommunikation krävs mellan namnnoden och datanoderna.
- Större block innebär färre block, vilket betyder att mindre metadata behövs för att beskriva systemet. Detta är viktigt då både GFS och HDFS lagrar all metadata i primärminnet, något som hade varit omöjligt med små block.

Det bör noteras att för traditionella filsystem avsedda för en enda hårddisk kan data bara lagras i multipler av blockstorleken. Om blockstorleken är 4KiB kommer till exempel en fil som är mindre det ändå uppta 4KiB på hårddisken. Detta gäller inte för HDFS [3].

Det är inte helt självklart hur blockstorleken i HDFS påverkar hårddiskens söktid. I *Hadoop – The Definitive Guide* [3] sägs så här om blockstorlek och söktid:

> *HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. If the block is large enough, the time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus, transferring a large file made of multiple blocks operates at the disk transfer rate.*

Det bör noteras att detta inte nämns som motiv till blockstorleken för GFS [8], men det finns en poäng i det. Varje block i HDFS lagras som en fil i det lokala filsystemet. Det finns ingen garanti för att en fil består av hårddiskblock i sekvens. Det skiljer sig mellan olika filsystem, men i praktiken är det svårt för ett filsystem att göra mer än att åtminstone försöka placera en fils hårddiskblock i närheten av varandra [9]. En enda fil kommer alltså behöva mer än en sökning för läsning, vilket leder till att blockstorleken i HDFS inte får lika stor effekt på söktiden som blockstorleken i det lokala filsystemet.

2.1.2 Replikering

Som det tidigare konstaterats är HDFS ett verktyg primärt avsett för big data, och som sådant måste hårdvarufel kunna hanteras. Ett av sätten som HDFS gör detta på är en

replikering - alla block replikeras på ett visst antal noder (tre stycken noder som standard) [3].

Denna replikering är så klart även relevant vid distribuerad exekvering. Att skicka block mellan noder är dyrt, så ett system för distribuerad exekvering måste prioritera att data bearbetas på samma noder som lagrar datan. Med replikering blir detta enklare då varje givet block har fler kandidater som kan bearbeta datan [3].

Att blocken replikeras kan betraktas som en av kostnaderna för distribuering. Det bör dock noteras att replikering är valfritt och det går utmärkt att använda HDFS även utan replikering.

2.3 MapReduce

MapReduce är en programmeringsmodell som tillåter distribuerad exekvering. Trots att MapReduce inte berörs direkt i rapporten följer här en kort introduktion för att ge ett kontext till hur distribuering har fungerat tidigare.

Modellen presenterades ursprungligen av Google och populariserades av Hadoop. Det bör noteras att MapReduce och Hadoops implementation av MapReduce är olika saker. När termen MapReduce används i rapporten är det programmeringsparadigmen som avses, när det är Hadoops implementation som avses kommer det specificeras som Hadoops MapReduce.

För att använda sig av MapReduce måste användaren definiera en implementation av dessa två metoder [2]:

```
Map({K1,V1}) →List[{K2, V2}]

Reduce(K2, List[V2]) →List[{K3,V3}]
```

MapReduce tar en lista av par (bestående av en nyckel och ett värde) och producerar, med hjälp av metoderna ovan, en ny lista av par. Här följer ett en förklaring och ett exempel på hur dessa två metoder används av MapReduce för att uppnå detta.

I det första steget appliceras mapfunktionen. För varje par returnerar mapfunktionen ett antal nya par. Tillsammans bildar alla dessa delresultat en enda lista av par. Exempel:

```
List[{nyckel1, 1}, {nyckel2, 2}, {nyckel1, 4}
```

blir efter mappning med funktionen `Map({K, V}) →{K, V * 2}`

```
List[{nyckel1, 2}, {nyckel2, 4}, {nyckel1, 8}
```

I det andra steget aggregeras värdena ihop baserat på nyckel, så att alla nycklar blir unika. De värden som har samma nyckel bildar en lista som blir nyckelns nya värde. Om resultatet av mappningen ovan aggregeras fås följande resultat:

```
List[{nyckel1, List[2, 8]}, {nyckel2, List[4]}]
```

I det tredje och sista steget appliceras reduceringsfunktionen för att reducera varje lista av värden till ett värde, det vill säga att varje nyckel efter reducering har ett enda värde. Om resultatet av aggregeringen ovan reduceras med funktionen `Reduce({K, V}) →{K, sum(V)}` (där `V` alltså är en lista av värden tillhörande nycklen `K`) fås följande slutgiltiga resultat:

```
List[{nyckel1, 10}, {nyckel2, 4}]
```

Den stora svagheten hos MapReduce är den väldigt låga abstraktionsnivån. Map och Reduce är visserligen högnivåoperationer, men att beskriva andra högnivåoperationer (till exempel filtrering) med enbart Map och Reduce är ointuivt.

Hadoops MapReduce har även ett annat mycket stort problem, som egentligen inte har med MapReduce som modell att göra. För icke triviala algoritmer krävs flera körningar av MapReduce-rutinen (mappning + aggregering + reducering) för implementering. Detta är egentligen inget problem, men Hadoops MapReduce kräver att indatan kommer från sekundärminnet. Detta betyder att för varje varv i MapReduce-rutinen måste all data läsas/skrivas till/från sekundärminnet, vilket så klart är en stor prestandakostnad [3].

2.4 Spark

Spark [16] är precis som Hadoops MapReduce ett verktyg för distribuerad exekvering. Svagheterna hos MapReduce och Hadoops MapReduce återfinns inte hos Spark. För det första data lagras i primärminnet under en hel algoritm, vilket ger Spark ett övertag i prestanda (särskilt vid iterativa analyser [10]).

För det andra uppnår Spark en avsevärt högre abstraktionsnivå genom att tillhandhålla ett brett API med en mängd olika högnivåoperationer istället för bara Map och Reduce. Dessa högnivåoperationer innefattar både Map och Reduce, så rent konceptuellt kan det sägas att Sparks API är ett superset av MapReduce: det går alldeles utmärkt att implementera en algoritm enligt MapReduce i Spark, men ett Spark-program är vanligtvis inte direkt översättbart till MapReduce [1].

2.4.1 API

Sparks API består primärt av två distribuerbara klasser, så kallade Resilient Distributed Datasets (RDD). Deras (icke distribuerbara) motsvarigheter i Javas standardbibliotek är Map och List (RDD-klasserna kommer därav senare att refereras till som Map-RDD och List-RDD). Metoderna på dessa klasser kan delas in i två typer: transformationer och handlingar [3].

- Transformationer beräknar inte resultatet direkt utan arbetet skjuts upp till resultatet faktiskt behövs (en teknik som brukar kallas *lazy evaluation*). Ett exempel på en transformation är `filter`, vars användningsområde är att filtrera bort oönskade element i ett RDD-objekt. Transformationer returnerar alltid ett nytt RDD-objekt, objektet som transformationen anropades på modifieras ej [3].
- Handlingar returnerar ett konkret värde, till exempel `count()`, som räknar antalet element i en samling. Det är när handlingar anropas som tidigare anropade transformationer appliceras [3].

Lazy evaluation innebär att tidskomplexiteten kan bli lägre än väntat. Om filtermetoden anropas två gånger innan en handling räcker det fortfarande att iterera samlingen en gång (då båda filtreringarna kan ske under samma iterering).

Ett vanligt exempelprogram inom distribuerad exekvering är ett program som beräknar antal förekomster av varje ord i en text. Med Hadoops MapReduce blir detta program över 50 rader, men med Spark är koden så liten att den får plats här nedanför. Största skälet till storleksskillnaden är att Sparks API använder sig av lambdafunktioner.

```
JavaSparkContext sc = new JavaSparkContext(new SparkConf());
sc
      .textFile("hdfs://...")
      .flatMap((s) -> Arrays.asList(s.split(" ")))
      .mapToPair((s) -> new Tuple2<String, Integer>(s, 1))
      .reduceByKey((a, b) -> a + b)
      .saveAsTextFile("hdfs://...");
```

2.5 Javaströmmar

I Java 1.8 introducerades ett strömpaket i form av `java.util.stream`. Klasserna är designade för parallell körning och strömklassens API är mycket likt RDD-klassernas API i Spark. Exempel på dessa likheter visas i tabellen nedan.

Spark	Streams
distinct	distinct
filter	Filter
map	map
flatMap	flatMap
take	limit
findFirst	first
reduce	reduce
reduceByKey	-
mapToPair	-

Utdrag ur API:erna som visar några av likheterna

Det finns dock några metoder i Spark som inte har motsvarigheter i Javaströmmar. Detta har att göra med att Spark har två distribuerbara klasser, List-RDD och Map-RDD. Strömklassen saknar de metoder som är relaterade till Map-RDD, men har dem som är relaterade till List-RDD. I tabellen ovan kan till exempel ses `mapToPair` , en metod som konverterar en List-RDD till en Map-RDD, inte har någon motsvarighet hos strömklassen.

Precis som i Sparks RDD's delas metoderna hos Javas strömmar in i transformationer och handlingar, med en viktig skillnad: efter att en handling har anropas går det inte längre att anropa varken transformationer eller handlingar på ström-objektet. Efter att en handling anropats resulterar alla påföljande anrop av handlingar och transformationer i att en exception kastas [6].

Det har funnits strömliknande klasser i Java även innan version 1.8, till exempel i form av `java.io.Reader`. Dessa har varit jämförelsevis primitiva och har inte tillåtit parallell

körning. När termen Javaströmmar eller strömmar används i denna rapport är det alltid `java.util.stream` som avses.

2.5 Relaterat arbete

Flera djupgående utvärderingar av Hadoops MapReduce har gjorts med olika metoder [11, 12]. Som konstaterats är MapReduce dock en mycket nischad model, och Hadoops implementation har ytterligare problem. Detta leder till att MapReduce har en begränsad relevans för lokal parallellisering - MapReduce bör inte användas om det inte är absolut nödvändigt. Dessa utvärderingar har därför begränsad relevans för denna rapport som specifikt berör användning av distribueringsverktyg i de situationer där det inte är nödvändigt.

Även av Spark har det gjorts utvärderingar, till exempel *Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark* [10] som berör prestandaskillnader mellan Spark och Hadoops MapReduce vid iterativa algoritmer.

3. Metod

Rapportens metodik består av tre delar:

- Utveckling av ett Sparkprogram
- Prestandatester av Spark, Javaströmmar och SQL
- Ett abstraktionslager som normaliserar lokal och distribuerad parallellisering

Sparkprogrammet utvecklades i syfte att undersöka hur Spark är att arbeta med, för att ge en förståelse för när distribuering är lämpligt, samt för att ha ett mer verkligt exempel att köra prestandatester på.

De prestandamätningar som gjorts hade fokus på hur HDFS och Spark kan användas med små dataset som inte kan klassas som big data.

3.1 Abstraktionslager för Spark och Javaströmmar

Då Spark ej är optimerat för lokal körning är det rimligt att strömmar kommer prestera bättre i den situationen. Som konstaterades tidigare i rapporten är API:erna för Spark och Javas strömmar mycket lika varandra. Det vore därför intressant att undersöka möjligheterna till att skapa ett abstraktionslager som döljer skillnaderna mellan lokal och distribuerad parallellisering.

Ett sådant abstraktionslager innebär i bästa fall att distribuering blir en ren konfigurationsfråga, istället för något som aktivt måste tas hänsyn till vid utveckling. Det innebär också att det blir trivialt att hantera problem som endast ibland är lämpade för distribuering (det vill säga att huruvida exekveringen skall ske distribuerat bestäms dynamiskt vid körningen).

Detta är relevant då det är en annan aspekt på rapportens problemformulering. Om det går att arbeta med distribueringsverkyg på detta sätt blir kostnaden i princip noll. Även om ett problem till en början är olämpligt att lösa med distribuering kan förutsättningarna ändras (kanske en analys som ändras så att den blir mer prestandaintensiv, eller ett dataset som växer okontrollerat).

3.1.1 Implementering

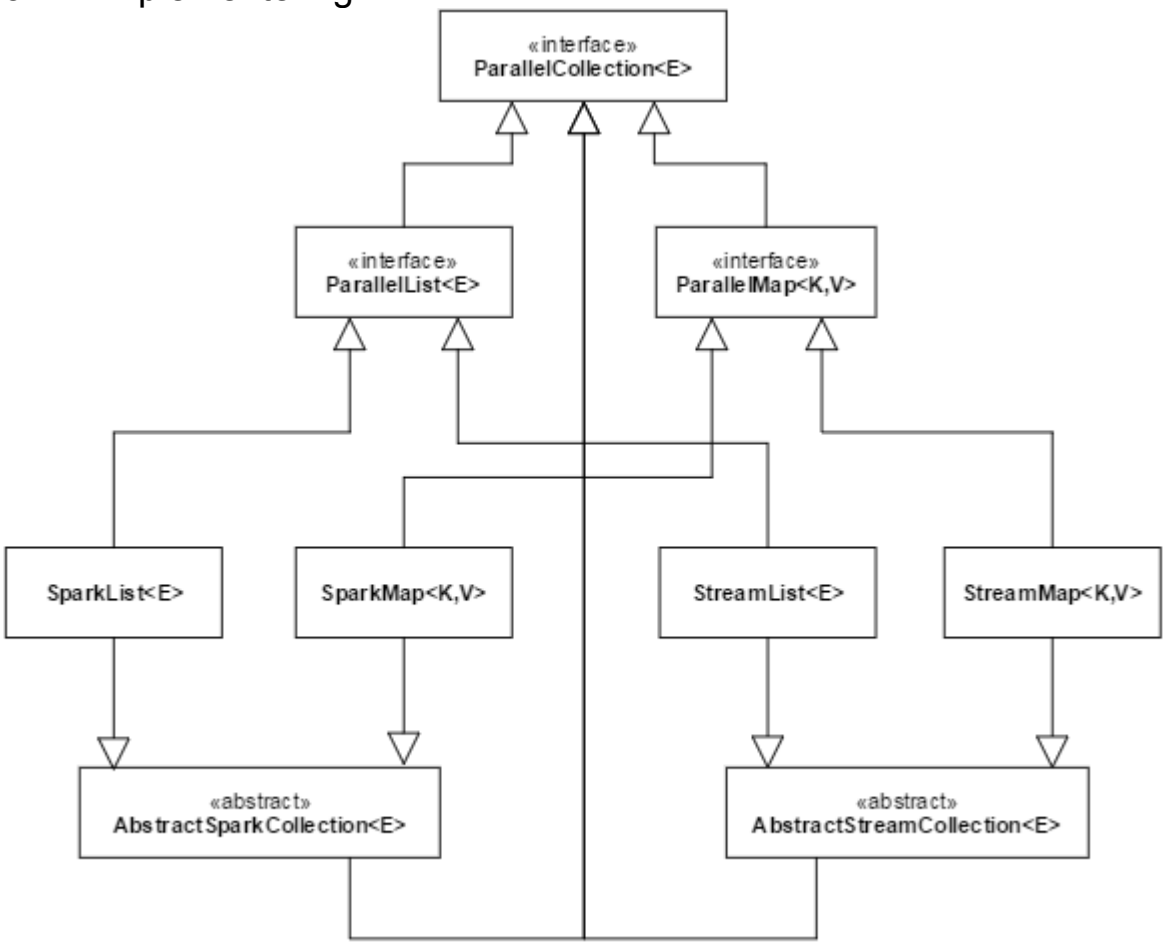


Diagram över några av klasserna i abstraktionslagret

Kärnan i abstraktionslagret är de två interfacen `ParallelList` och `ParallelMap`, som direkt motsvarar List-RDD och Map-RDD. Metoderna i dessa interfacen är baserade på metoderna i motsvarande RDD-klass (alla metoder kunde dock inte implementeras på grund av tidsbrist). Abstraktionslagret tillhandhåller två stycken implementationer för varje interface, en som använder Spark och en som använder strömmar.

De implementationer som använde Spark var triviala att implementera då alla metoderna hade direkta motsvarigheter i RDD-klasserna. Även för strömimplementationerna är de flesta metoder enkla att implementera, oftast så enkla att de ryms på en enda rad (exemplet som visas är implementationerna för `ParallelList#filter`):

```
public StreamList<T> filter(Function<T, Boolean> predicate) {
      return new StreamList<>(getStream().filter(predicate::apply));
}

public SparkList<T> filter(Function<T, Boolean> predicate) {
      return new SparkList<>(_rdd.filter(predicate::apply));
}
```

Det finns dock fall där Javas strömklass inte har någon metod som direkt motsvarar metoden som skall implementeras. Ett sådant exempel är `ParallelList#reduceByKey`, som fyller samma funktion som aggregeringssteget tillsammans med reduceringssteget i

MapReduce. Metoden tar en reduceringsfunktion som argument och använder den för att reducera alla värden med samma nyckel till ett enda värde. Strömimplementationen innehåller två steg. Först sker en iterering av alla element i samlingen. I denna iterering reduceras succesivt alla värden med samma nyckel och resultatet lagras i en `HashMap`. I det andra steget omvandlas denna `HashMap` till en ny `StreamMap`.

Denna implementation använder sig alltså inte av lazy evaluation, till skillnad från Sparkimplementationen. Detta är något som ett mer sofistikerat abstraktionslager hade kunnat göra annorlunda, men även denna implementation visade sig prestera bra.

Det största problemet med strömimplementationerna är att Javas strömmar inte stödjer att mer än en handling anropas på samma ström. Om flera handlingar anropas kastas en exception. För RDD-klasserna finns ingen motsvarande begränsning. Detta löstes på ett primitivt sätt. Om användaren av API:et vill anropa handlingar eller transformationer på en `ParallelCollection` efter att den en handling redan har anropats så måste metoden `terminate()` anropas innan första handlingen anropas. Metoden kopierar strömmens innehåll till en lista som sedan kan omvandlas till en ström vid behov. På så sätt fås en ström som kan återskapas.

Denna lösning har två stora problem. För det första innebär det att strömmens innehåll måste rymmas i minnet när `terminate()` anropas och för det andra innebär det att hela strömmen måste itereras en extra gång. Dessa problem har dock inget med konceptet i sig att göra, utan beror på Javas implementation av strömmarna. Det är även möjligt att det går att hantera situationen på ett bättre sätt än ovan och ändå använda Javas strömmar.

För Sparkimplementationerna är inget av detta nödvändigt då det som sagt inte finns någon motsvarande begränsning i RDD-klasserna. Spark har dock en funktion som motsvarar beteendet hos `terminate()` för strömimplementationerna. Metoden `cache()i` RDD-klasserna säger till att värdena skall sparas i minnet *efterhand* som de beräknas, om de får plats. Det betyder att ingen extra iteration behövs till skillnad från strömimplementation av `terminate()`.

En sista notering bör göras gällande namnet `ParallelMap`, som lätt kan vara något missvisande. En `Map` (specifikt en `HashMap`) är något som i vanliga fall kännetecknas av att kunna läsa ett godtyckligt element snabbt (O(1)). Detta är inte något som gäller för `ParallelMap` – enda sättet att läsa ett godtyckligt element är att iterera igenom hela samlingen tills det hittas (det vill säga O(n)).

Detta har att göra med att `ParallelMap` (och Map-RDD) inte är designade med tanken att element skall hanteras enskilt. Syftet med klasserna är att bearbeta *alla* element så snabbt som möjligt.

3.2 Exempelprogram - textgenerering

För att ha ett konkret program att köra prestandatester på utvecklades ett exempelprogram som genererar ett textflöde baserat på ett dataset av existerande text.

Programmet kan delas in i två delar. Först måste en databas med ord skapas, och sen måste orden väljas baserat på vilket ord som är mest lämpat.

För att avgöra lämplighet av ett ord sparas de två tidigare orden i en ordföljd som en hash. Det vill säg om indata till algoritmen är meningen “The quick brown fox jumps over the lazy dog” kommer databasen av ord se ut så här (där `h` är en hashfunktion):

Hash	Ord
h(the quick)	brown
h(quick brown)	fox
h(brown fox)	jumps
h(fox jumps)	over
h(jumps over)	the
h(over the)	lazy
h(the lazy)	dog

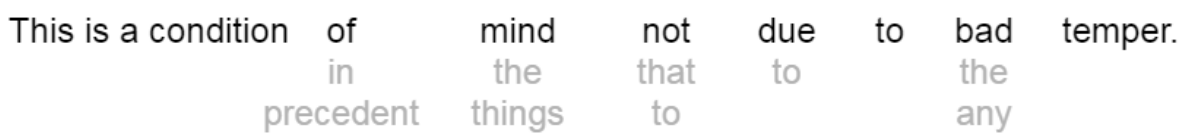
Databas genererad från en mening

Databasen håller även koll på hur många gånger varje unikt par av `{hash, ord}` har förekommit. Om även meningen “A fox jumps over me” skulle skickas in i algoritmen blir resultatet detta:

Hash	Ord	Förekomster
h(the quick)	brown	1
h(quick brown)	fox	1
h(brown fox)	jumps	1
h(fox jumps)	over	2
h(jumps over)	the	1
h(over the)	lazy	1
h(the lazy)	dog	1
h(a fox)	jumps	1
h(jumps over)	me	1

Databas genererad från två meningar. Skillnader från föregående markerade i grått

Denna data kan sedan användas för att ge förslag på nästa ord till en existerande text. Algoritmen väljer helt enkelt ut de N mest förekommande raderna med en viss hash och väljer ett av orden slumpmässigt. Om den befintliga texten är “jumps over” kommer algoritmen kunna välja mellan “me” och “the” om exempeldatabasen används. Så här kan det se ut om en stor orddatabas används, där “this is a condition” har använts som starttext (gråa ord är kandidater som valts bort av algoritmen):



Nedan följer några exempel på texter som programmet genererat:

> *"One of the great city, and the young fisherman saw a man who had been a great deal of good. And now I am not going to be a great deal of good."*
>
> *"It becomes grotesque, and not the slightest noise. Have you no end of my life."*
>
> *"No one has ever given or received this token of affection."*
>
> *"You wouldnt have me take thy soul? I am bidden to take command."*

3.3 Prestanda

I detta kapitel undersöks hur exekveringstiden varierar när fler noder läggs till, hur Spark står sig gentemot lokal parallellisering samt vad som händer med responstiden i SQL när antalet tabellrader ökar.

Prestandamätningarna gjordes på fyra olika datorer som visas nedan. Läshastigheten för D1, D2 och D3 var ~80MiB/s och ~500MiB/s för D4. Alla datorerna använde ext4 som filsystem.

#	Model	Operativsystem	Processor
D1	HP EliteBook 8730w	Ubuntu 16.04	Intel Duo T9600 2.80GHz
D2	HP EliteBook 8730w	Ubuntu 16.04	Intel Duo T9400 2.53GHz
D3	HP EliteBook 8730w	Ubuntu 16.04	Intel Duo T9600 2.80GHz
D4	Lenovo Thinkpad x250	Arch Linux	Intel Core i3-5010U 2.10GHz

Beskrivning av klustrets noder

Som kan ses hade D2 något sämre processor än de andra, vilket så klart påverkar korrektheten i de mätningar som gjorts. D4 användes som namenode samt som schemaläggare av övriga noder, därav är dess lägre prestanda (för processorn) och annorlunda konfiguering ej något som bör påverka resultatet negativt.

3.3.1 Schackanalys

Det tycks finnas en allmän uppfattning om att verktyg avsedda för att arbeta med stora datamängder aldrig är lämpliga för små datamängder på grund av kostnader vid distribuering. Då kostnaderna definitivt finns är detta inte helt osant, men kostnaderna överdrivs ofta.

Ett exempel på detta är blogginlägget *Command-line tools can be 235x faster than your Hadoop cluster*, där en implementation av en schackanalys presenteras. Analysen bearbetar ett dataset av schackspel och beräknar hur många matcher svart vunnit, hur många vitt vunnit samt hur många som slutat i remi. Implementationen använder sig av terminalverktyget mawk. Prestandan av denna implementation jämförs med ett tidigare blogginlägg där en implementation med Hadoops MapReduce gjorts med resultatet att den nya implementationen är 235 gånger snabbare. Slutsatsen är tydlig: kostnaden för distribuering vid små datamängder är enorm [5].

Detta blogginlägg kan lätt framstå som anekdotiskt och irrelevant, men det är en del av ett större fenomen där distribuering i allmänhet och Hadoop i synnerhet anses irrelevant för små datamängder. Att det ovan nämnda blogginlägget har valts som exempel är ingen slump. På forumet Reddits underforum om programmering [13] (med i skrivande stund över 660000 registrerade läsare) är detta blogginlägg två av de tre mest diskuterade länkar om Hadoop [14]. Den tredje länken är ett blogginlägg vars titel har ett liknande budskap: *Don't use Hadoop - your data isn't that big*. Det bör påpekas att båda dessa blogginlägg är flera år gamla, men budskapet tycks leva kvar ändå.

För att visa hur kostnaden vid distribuering faktiskt ser ut kommer i rapporten att redovisas en prestandajämförelse för samma analys med tre olika metoder: mawk (exakt samma kod som användes i blogginlägget), Javaströmmar samt Spark. Hela datasetet är ~6.6GiB. Mätningarna gjordes på D4.

Nedan följer koden som användes för Javaströmmar och Spark (abstraktionslagret användes, så därav att koden är samma för båda). `Provider` är ett interface som används för att skapa instanser av `ParallelMap` och `ParallelList` från data i filsystemet. Beroende på om Spark eller Javaströmmar används så används olika implementationer av `Provider`.

```
lines = provider.textfile("path/to/files");
lines = lines.filter((line) -> line.startsWith("[Result"));

ParallelMap<Winner, Integer> results = lines.mapToPair((line) -> {
      char khar = line.charAt(11);
      if (khar == '0') return new Tuple2<>(Winner.White, 1);
      if (khar == '1') return new Tuple2<>(Winner.Black, 1);
      if (khar == '2') return new Tuple2<>(Winner.Draw, 1);
      return new Tuple2<>(Winner.Invalid, 1);
});

results = results.reduceByKey(0, (a, b) -> a + b);
results.collect().forEach(System.out::println);
```

Koden för mawk följer här:

```
find . -type f -name '*.pgn' -print0 | xargs -0 -n4 -P4 mawk
'/Result/ { split($0, a, "-"); res = substr(a[1], length(a[1]), 1);
if (res == 1) white++; if (res == 0) black++; if (res == 2) draw++
} END { print white+black+draw, white, black, draw }' | mawk
'{games += $1; white += $2; black += $3; draw += $4; } END { print
games, white, black, draw }'
```

Information om hur datasetet ser ut kan läsas i den ursprungliga källan [5].

3.3.2 Textgenerering

I syfte att undersöka prestandan hos Spark för ett verkligt användningsområde användes verktyget för textgenerering. Endast skapandet av termerna mättes. Cirka 170MiB slumpmässigt utvald engelskspråkig text från Project Gutenberg [15] användes som dataset för att bygga orddatabasen.

Varierande uppdelningsbarhet testades genom att variera antal filer som datasetet delades upp på. Detta är ett väldigt grovt sätt att mäta på, men det funkar som approximation då Spark tycks föredra att en och samma fil inte bearbetas av flera noder.

Det gjordes även mätningar med varierande antal noder i klustret. Som noterats är dessa värden inte helt pålitliga på grund av att den ena noden (D2) hade något sämre processor. Dessa värden jämfördes även med en körning av koden med strömmar och en körning med Spark i lokalt läge.

3.3.3 Random-read

Random-read operationer, att hitta ett specifikt värde bland många andra, är något som generellt sett är väldigt olämpligt att distribuera. Ett random-read verktyg utnyttjar primärminnet för att lagra en sorterad representation (till exempel index i SQL) av alla värden som sökningen skall ske bland. Läsning från primärminnet, till skillnad från sekundärminnet, är så tidseffektivt att kostnaden för distribuering är för hög för att det skall vara lämpligt.

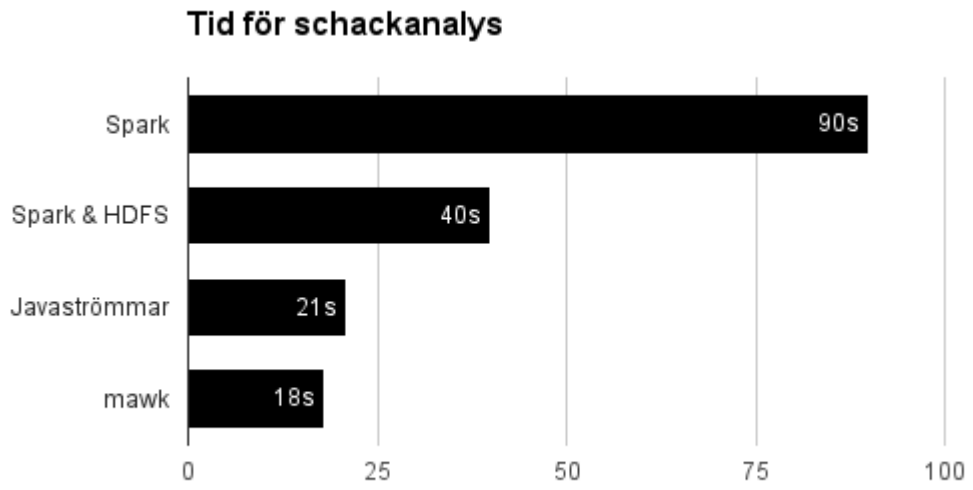
Däremot är primärminnet väldigt begränsat i storlek. För random-read operationer med mycket stora dataset är det inte möjligt att lagra all data som behövs i det lokala primärminnet. När denna situation inträffar kan det vara lämpligt att använda en distribuerad databas, som i praktiken inte har någon övre gräns på hur mycket primärminne som kan finnas tillgängligt i klustret.

För att visa hur läshastigheten för random-read påverkas när primärminnet inte räcker till görs några prestandatest. MariaDB användes som databas med InnoDB som databasmotor. En väldigt låg storlek på primärminnet användes för att det skulle bli enklare att testa: 16MiB. Testerna kördes på D4.

4. Resultat

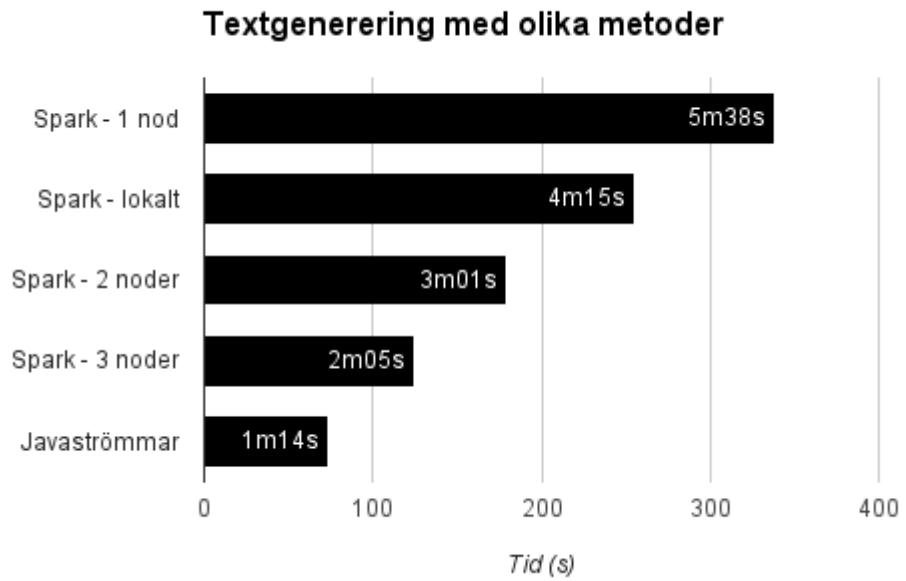
All rå mätdata för prestandatesterna presenteras i tabellform i appendix.

4.1 Schackanalys



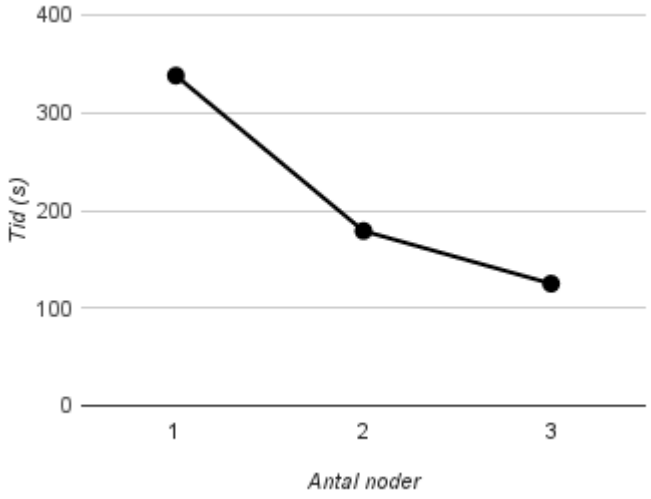
Det bör noteras att koden för *Spark & HDFS* och *Javaströmmar* är identisk då abstraktionslagret användes. Spark kördes i lokalt läge.

4.1 Textgenerering



Det bör noteras att strömmetoden kördes med 7GiB tillgängligt primärminne medan noderna i klustret hade tillgång till cirka 3GiB vardera. Strömmetoden är inte körbar alls med 3GiB primärminne, på grund av att abstraktionslagret utgår från att allt kommer få plats i primärminnet när strömmar används. Ett mer sofistikerat abstraktionslager hade kunnat hantera detta, till exempel genom att spilla data till sekundärminnet (vilket är vad Spark gör).

Då Spark kördes i lokalt läge gjordes det med lika mycket tillgängligt primärminne som när strömmetoden kördes, och behövde trotts detta ~2.4 gånger så mycket tid.



Som kan ses minskar tiden mindre då den andra noden läggs till. Detta är rimligt då varje ny nod bidrar med mindre arbetskraft relativt till vad som redan finns i klustret. För att få en grov uppfattning om hur exekveringstiden kommer utvecklas då fler noder läggs till kan med bakgrund av detta några beräkningar göras.

Det är rimligt att allt arbetet inte går att distribuera. I teorin bör exekveringsdelen kunna delas in i en relativt konstant del (t_k) och en distribuerbar del (t_d). Det går då att ställa upp sambandet $t_k + t_d / n = t_{TOT}$, där n är antalet noder och t_{TOT} är den totala exekveringstiden. Mätvärdena ger då följande värden för t_k:

$$R1 : t_k + t_d / 1 = 338$$
$$R2 : t_k + t_d / 2 = 179$$

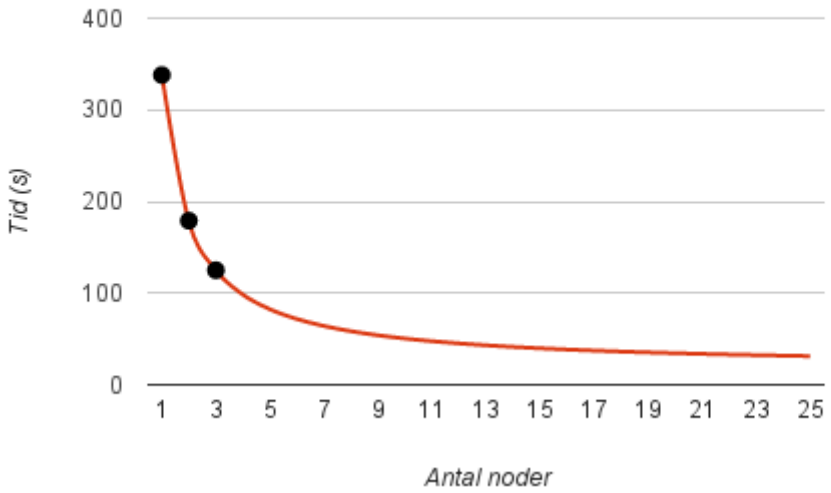
$$R3 : t_k + t_d / 3 = 125$$

$$R1 \vee R2 \Rightarrow t_k = 20$$
$$R2 \vee R3 \Rightarrow t_k = 17$$

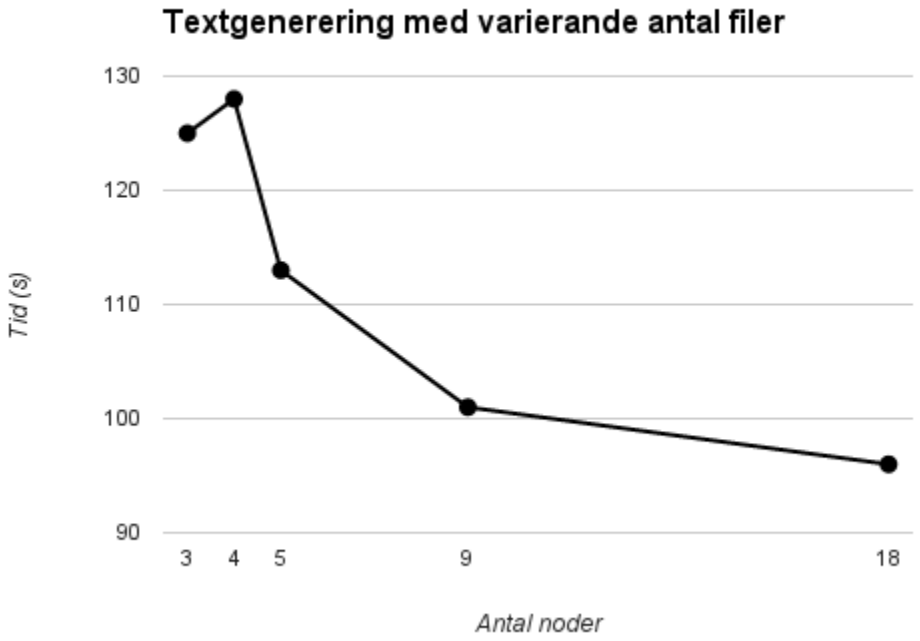
Som kan ses är t_k inte riktigt konstant utan minskar något när antalet noder ökar. Om det antas att detta inte är en trend utan bara varians kan vi sätta upp följande ekvation (18.5 är medelvärdet av de olika t_k-värdena):

$$t_{TOT} = 18.5 + (338 - 18.5) / n$$

Nedan visas en graf som använder denna ekvation som trendlinje för mätvärdena:

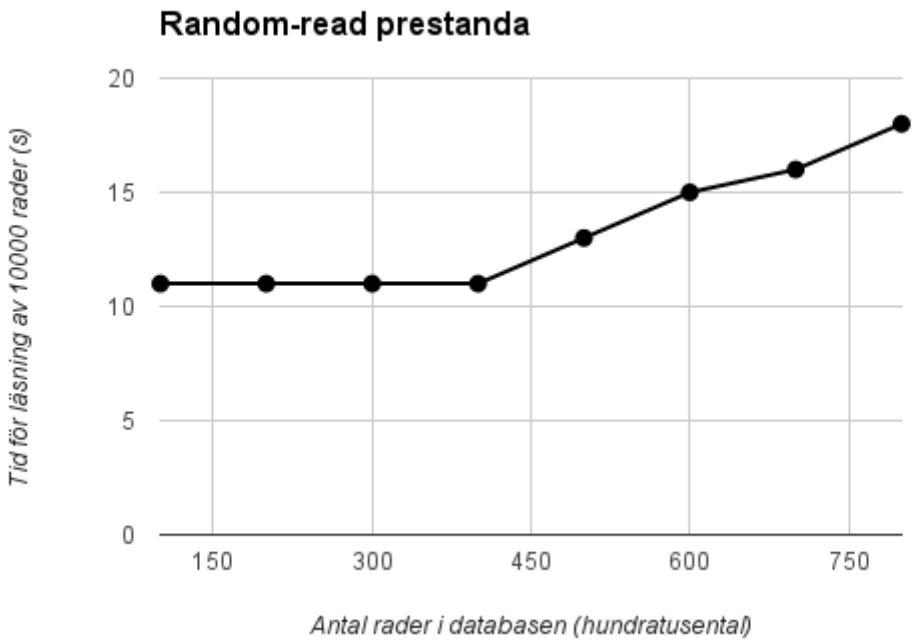


Det kan slutligen konstateras att mätvärdet för strömmar (71 sekunder) enligt denna ekvation kommer uppnås vid sex stycken noder, ty `n = (338 - 18.5) / (71 - 18.5)` ≈ 6.



Observera att y-axeln ej börjar på noll. Det första värdet, för två noder, är utelämnad på grund av att det är så högt att grafen blir svårläst. Exekveringstiden för två noder är 350 sekunder, nästan tre gånger så långt tid som för tre noder.

4.1 SQL



Vid det femte mätvärdet upptog alla index ~18.5MiB. Då det är högre än det tillgängliga primärminnet (16MiB) började lästiden öka. Så klart bör lästiden öka även när allt får plats i primärminnet, men vid så här små radantal är det ej märkbart.

Det bör noteras att det i riktiga scenarier krävs extrema mängder data för att indexen inte skall rymmas i primärminnet. Om det antas att indexen kommer kräva lika utrymme som i testet ovan (~35 bytes per rad) och kommer varje GiB primärminne rymma indexen till 28 miljoner databasrader. Så klart kommer dock ett verkligt scenario ha en mer komplex databas med flera tabeller och index, så det måste avgöras för varje enskilt fall om en distribuerad databas är lämpligt.

Ytterligare en notering är att testet ovan antar att alla rader i databastabellen är lika sannolika att läsas. I väldigt många situationer (till exempel chattjänster) kommer nyare rader läsas oftare, medans väldigt gamla rader nästan aldrig behöver läsas. Om indexet som används är ett ordnat id (det vill säga att nyare rader har ett högre eller lägre id än äldre rader) går det att garantera att de X nyaste eller äldsta raderna (beroende på konfigurering) alltid kommer finnas i primärminnet (då de index som lagras i primärminnet när inte alla får plats väljs baserat på deras sorterade ordning).

5. Diskussion

Utifrån prestandamätningarna och den teoretiska bakgrunden kan det konstateras att de primära faktorerna som avgör huruvida ett problem är lämpligt för distribuering är dessa:

- Uppdelningsbarhet
- Indatastorlek
- Exekveringstid

Dessa definieras och förklaras i detalj nedan. Om dessa faktorer förstås, går det också att avgöra när distribuering är lämpligt. Noterbart är att ingen av dessa faktorer kräver speciellt stora datamängder för att de skall vara relevanta.

5.1 Uppdelningsbarhet

Ett problems uppdelningsbarhet är antalet arbetsuppgifter som kan bearbetas parallellt. Naturligtvis kan uppdelningsbarheten variera under körningen av ett och samma program, vilket måste vägas in i bedömningen.

Uppdelningsbarheten är ett mått på parallelliseringens potential. Det följer naturligt att ett problem som bara kan delas upp i N delar inte kan distribueras på N + 1 noder på ett meningsfullt sätt. Detta syns tydligt i prestandamätningarna.

Som prestandamätningarna också har visat är uppdelningsbarheten relevant även när den är högre än antalet noder. Skillnaden är dock mindre än när uppdelningsbarheten är mindre än antalet noder. Tre delar tog ~35% av tiden för två delar, medan arton delar tog 77% av tiden för tre noder. Mellan nio och arton noder är skillnaden marginell. Det är alltså tydligt att prestandavinsten faller av snabbt, även om det kommer finnas en mätbar prestandavinst ändå upp till cirka 200 noder.

Det är intressant är att exekveringstiden *ökar* när antalet delar ökar från tre till fyra. Förklaringen kan tänkas vara att fyra stycken delar är särskilt svårt att distribuera på tre noder. Detta bör gälla för alla situationer med N noder och N + 1 delar, men om N inte är mycket lågt kommer det antagligen inte märkas.

Så klart är en viss uppdelningsbarhet nödvändigt även vid lokal parallellisering, men inte till samma grad. Lokalt är graden av parallellisering begränsad till antalet processorkärnor, som i praktiken är extremt begränsat. För distribuering å andra sidan är parallelliseringsgraden bara begränsad av antalet noder, som kan vara så högt som flera tusen.

5.2 Indatastorlek

Läsning av data från sekundärminnet är en relativt dyr operation, i vissa fall (till exempel vid random-read operationer) så pass dyr att det måste undvikas till varje pris. I många fall går det att undvika läsning från sekundärminnet genom lagring i primärminnet, men den mindre storleken på primärminnet skapar begränsningar.

Distribuering tacklar detta problem på två sätt. Dels genom att ett kluster gemensamt har tillgång till avsevärt mer primärminne än en enskild nod kan ha, vilket minskar behovet av läsning från sekundärminnet. I de fall då läsning från sekundärminnet är oundvikligt

hjälper distribuering också till: om datan läses in parallellt på alla noder uppnås en högre läshastighet än vad en ensam nod kan uppnå.

5.3 Exekveringstid

De kostnader som finns vid distribuering innebär att för små problem som kan köras lokalt på väldigt kort tid är det omöjligt att optimera genom distribuering. Eventuella vinster av den ökade parallelliseringen är mindre än kostnaderna som uppstår som resultat av kommunikation mellan noder och dylikt.

Kostnaden vid distribuering bestämdes till 18.5 sekunder vid prestandatestet för textgenerering. Kostnaden är knappast den samma oavsett situation, utan kan tänkas bero på till exempel antal noder i klustret. Det ger likväl en vis bild av vilken storleksordning som distribueringens kostnad befinner sig i. Det kan tänkas att om exekveringstiden för en lokal implementation är mindre än detta så är det mycket osannolikt att distribuering kan tillämpas.

Slutligen bör det poängteras att Spark presterar mycket dåligt vid lokal körning, men att det inte har att göra med API-modellen. Merparten av Sparks API är trivialt att implementera på ett prestandaeffektivt sätt för lokal körning. Sparks prestanda vid lokal körning begränsas snarare av att Spark ej är avsett att användas på det sättet, möjligheten till lokal körning finns endast för att kunna arbeta med Spark på ett smidigt sätt.

6. Slutsats

Rapporten har visat hur verktygen Spark och HDFS kan, trotts att de är designade med stora datamängder i åtanke, vara effektiva även för att bearbeta relativt små dataset. Prestandatesterna har visat att det visserligen existerar stora kostnader vid distribuering, men då prestandavinsten vid tilläggning av noder är så pass hög spelar det inte så mycket roll.

Det har i rapporten även visats hur distribuering inte längre är begränsat till nischade programmeringsmodeler som MapReduce. Sparks API ligger på en så pass hög abstraktionsnivå att det i grund och botten inte innebär någon begränsning alls - liknande API:er finns implementerade i områden som inte alls har med distribuering att göra. Att motsvarande skulle gälla för MapReduce vore otänkbart.

På grund av detta kan det också konstateras att det abstraktionslager som presenteras i rapporten både är möjligt och relevant. När "små" problem distribueras kommer det ofta inträffa situationer där det *ibland* är önskvärt att distribuera. Ett abstraktionslager likt det som presenteras i rapporten löser detta problem.

I prestandamätningarna konstateras att kostnaden för distribuering av textgenereringen är 18.5 sekunder. Detta är så klart ett väldigt ungefärligt värde, men det ger en viss insikt i när distribuering är lämpligt. I de fall där en lokal implementation har en exekveringstid på under 10 sekunder kan distribuering uteslutas direkt.

Detta värde är dock kostnaden vid ett oändligt antal noder. Kostnaden vid en enda nod är inte samma. Vid textgenereringen är kostnaden hela tre minuter, men vid schackanalysen är kostnaden bara cirka tjugo sekunder. Det är tydligt att det finns en stor varians i kostnaden. Variansen kan antas vara större när bara en nod används.

Slutligen kan det konstateras att det faktiskt finns områden där distribuering av små datamängder förmodligen alltid kommer vara en dålig idé. Ett exempel på detta är till exempel alla program som primärt arbetar i primärminnet med datamängder som är billiga att rymma i primärminnet på en ensam dator. Random-read är ett exempel på en sådan situation.

6.1 Vidare arbete

Det abstraktionslager som presenterats i rapporten är väldigt rudimentärt. Detta är ett relativt nytt område då de tidigare modellerna för distribuerad körning (till exempel MapReduce) varit ointuitiva så till den grad att det inte är lämpligt att använda dem när det går att undvika. Spark å andra sidan är lättare att arbeta med och är dessutom väldigt likt existerade API:et för lokal parallellisering (till exempel Javaströmmar), vilket gör denna typen av abstraktionslager både användbarare och enklare att implementera.

Ett ordentligt designat abstraktionslager skulle till stor del kunna eliminera behovet av "tänkta distribuerat". Om samma paradigmer som används för distribuerad parallellisering kan användas för lokal paradigmer blir distribuering bara en ren konfigurationsfråga.

De prestandatester som redovisas i rapporten är väldigt begränsade. För att ge ett mer användbart resultat hade ett större kluster behövts. Dessutom skedde ingen egentlig testning av HDFS. Visserligen användes HDFS i testerna, men en undersökning av hur prestandan i HDFS skiljer sig mot lokala filsystem hade varit relevant. Detta görs delvis i schackanalys, men bara med en enda nod.

Referenser

[1] ZAHARIA, Matei, et al.
Spark: Cluster Computing with Working Sets
HotCloud, 2010, 10: 10-10

[2] LÄMMEL, Ralf.
Google's MapReduce programming model—Revisited
Science of computer programming, 2008, 70.1: 1-30.

[3] WHITE, Tom.
Hadoop: The definitive guide. 4th edition
"O'Reilly Media, Inc.", 2015.

[4] BORTHAKUR, Dhruba, et al.
Apache Hadoop goes realtime at Facebook
In:Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011. p. 1071-1080.

[5] DRAKE, Adam.
Command line tools can be 235x faster than your hadoop cluster
http://aadrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html
(Hämtad 15 Maj 2015)

[6] Package java.util.stream Description
https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html
(Hämtad 15 Maj 2015)

[7] Default cluster size for NTFS, FAT, and exFAT
https://support.microsoft.com/en-us/kb/140365
(Hämtad 15 Maj 2015)

[8] GHEMAWAT, Sanjay; GOBIOFF, Howard; LEUNG, Shun-Tak.
The Google file system
In: ACM SIGOPS operating systems review. ACM, 2003. p. 29-43.

[9] CARRIER, Brian.
Ext2 and Ext3 concepts and analysis
Computer Security Journal, v 21, 2005, n 3, p 3060

[10] GU, Lei; LI, Huan.
Memory or time: Performance evaluation for iterative operation on hadoop and spark.
In: High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on. IEEE, 2013. p. 721-727.

[11] JIANG, Dawei, et al.
The performance of mapreduce: An in-depth study.
Proceedings of the VLDB Endowment, 2010, 3.1-2: 472-483.

[12] LEE, Kyong-Ha, et al.
Parallel data processing with MapReduce: a survey.
AcM sIGMoD Record, 2012, 40.4: 11-20.

[13] https://www.reddit.com/r/programming
(Hämtad 23 Maj 2016)

[14] https://www.reddit.com/r/programming/search?q=hadoop&sort=top&restrict_sr=on&t=all
(Hämtad 23 Maj 2016)

[15] http://www.gutenberg.org/

[16] https://spark.apache.org/

[17] http://hadoop.apache.org/

Appendix

Antal filer	Tid körning 1 (*s*)	Tid körning 2 (*s*)	Tid körning 3 (*s*)	Tid genomsnitt (*s*)
2	353	349	350	350
3	119	135	121	125
4	135	125	126	128
5	105	115	120	113
9	97	99	107	101
18	96	95	99	96

Textgenerering med varierande antal filer

Antal rader	Tid läsning (*s*)	Tid läsning av sista (*s*)	Total storlek av index (*MiB*)
100	11	11	4.5
200	11	11	7.5
300	11	11	11.5
400	11	11	14.5
500	13	11	18.5
600	15	11	21.5
700	16	11	25.5
800	18	11	28.5

SQL med varierande antal tabellrader

Metod	Tid körning 1 (*s*)	Tid körning 2 (*s*)	Tid körning 3 (*s*)	Tid genomsnitt (*s*)
Spark - 1 nod	305	330	380	338
Spark - lokalt	257	260	249	255
Spark - 2 noder	181	209	149	179
Spark - 3 noder	119	135	121	125
Javaströmmar	71	85	67	74

Textgenerering med olika noder samt med strömmar