

**مقارنة هادوب وسبارك من ناحية  
التسامح مع الأعطال وتقديم البحث من خلال تصميم موقع الكتروني**

الصفحات	العنوان	الرقم
	<b>الفصل الأول</b>	
4-3	منصة عمل هادوب ومكوناتها	1
5-4	نظام ملفات هادوب الموزع ومزاياه	2
6	مكونات نظام ملفات هادوب الموزع	3
7	التخزين المؤقت للكتل	4
7	الوضع الآمن	5
7	بيانات نظام الملفات الوصفية	6
8-7	مقارنة MapReduce مع YARN	7
10-9	تدفق البيانات Data Flow	8
12-11	كيفية عمل ماب ريديوس في هادوب	9
	<b>الفصل الثاني</b>	
14	التسامح مع الفشل في نظام ملفات هادوب الموزع	10
16-15	فشل العقدة الرئيسية والتسامح معه	11
16	تجاوز الفشل والمبادرة	12
17	فشل عقدة البيانات والتسامح معه	13
19-18	تكرار كتل البيانات	14
21-20-19	التسامح مع الفشل في هادوب ماب ريديوس	15
	<b>الفصل الثالث</b>	
23-22	مكدس أباتشي سبارك الموحد ومكوناته	16
24	طبقات تخزين البيانات في سبارك	17
25	التنفيذ الموزع في سبارك	18
27-26	مكونات عنقود سبارك	19
29-28	Spark on YARN	20

الصفحات	العنوان	الرقم
	<b>الفصل الرابع</b>	
30	التسامح مع الأعطال في سبارك	21
30	التسامح مع الفشل الذي تقدمه	22
31	نقاط التفتيش	23
32	التسامح مع فشل المشغل	24
32	التسامح مع فشل العقدة العاملة	25
32	التسامح مع فشل المنفذ	26
33	ضمانات المعالجة	27
	<b>الفصل الخامس</b>	
	<b>القسم العملي</b>	
34	مواصفات جهاز الحاسوب المستخدم	28
34	خطوات التنفيذ	29
37-36-35	تثبيت عنقود هادوب	30
39-38	تثبيت HiBench على العقدة الرئيسية وربطه مع هادوب	31
40	اختيار مجموعة من أعباء الحمل	32
41	معايير الأداء المدعومة والمستخدمة من قبل HiBench :	33
42-41	منهجية البحث المستخدمة في تنفيذ التجارب	34
44-43	تشغيل حمل العمل الخاص ببرنامج WordCount	35
47-46-45	اختيار مجموعة من حالات الفشل الممكنة في كلا المنصتين	36
48	تشغيل حمل العمل الخاص ببرنامج WordCount مع فشل مدير التطبيق MRAppMaster	37
49	تشغيل حمل العمل الخاص ببرنامج WordCount مع فشل التحطط Node Crash Failure	38
50	تثبيت HiBench على العقدة الرئيسية وربطه مع هادوب	39
51	تنفيذ وتقدير أداء سبارك عند أعباء الحمل المختلفة	39
52	تشغيل سبارك لحمل العمل الخاص ببرنامج WordCount بدون فشل	40
54-53	تنفيذ وتقدير أداء سبارك عند أعباء الحمل المختلفة (مع حقن فشل) وأنواعه:	41
	<b>الفصل السادس</b>	
56	زمن التنفيذ وزمن تجاوز الفشل (ثانية)	42
57	فشل مدير العقدة	43
57	فشل مدير التطبيق	44
58	فشل تحطم عقدة بيانات في هادوب	45
58	فشل Executors	45

<b>59-58</b>	<b>فشل التحطّم في سبارك</b>	<b>46</b>
<b>59</b>	<b>النتائج والتوصيات</b>	<b>47</b>

**قائمة الاختصارات:**

HPC	High Performance Computing
MPI	Message Passing Interface
API	Application Program Interfaces
RDMS	Relational Database Management System
SQL	Structured Query Language
SAN	Storage Area Network
HDFS	Hadoop Distributed File System
POSIX	Portable Operating System Interface
NFS	Network File System
MPP	Massively parallel processing
RAID	Redundant Array of Independent Disks
CLI	Command Line Interface
JVM	Java Virtual Machine
YARN	Yet Another Resource Negotiator
LRU	Least Recently Used

UDF	User-defined Functions
UDA	User-defined Aggregate
UDTF	User-defined Table Function
PDSH	Parallel Distributed Shell
QJM	Quorum Journal Manager
RDD	Resilient Distributed Data-set
ML	Machine Learning
LRU	Least Recently Used
DAG	Directed Acyclic Graph



## الفصل الأول

### منصة العمل هادوب

#### Hadoop Framework

##### 1-1 مقدمة:

إنَّ أبانتشي هادوب أداة مفتوحة المصدر من Apache Software Foundation -ASF ويعتبر أشهر تقنيات إدارة البيانات الضخمة حيث يخزن ويعالج البيانات المختلفة مما يمكن الشركات المُقادمة بالبيانات Data Driven Companies من استخلاص القيمة الكاملة لكل بياناتهم، وهو بيئة برمجية لكتابه وتنفيذ التطبيقات الموزعة التي تعالج كميات ضخمة من البيانات، في الوقت الحاضر هادوب هو جزء مهم من البنية التحتية للمعالجة في كثير من شركات الإنترنت مثل [3] LinkedIn.

يوفِر هادوب إطاراً فعَالاً لتشغيل المهام بالتوالي على عدة عقد متصلة عبر الشبكة المحلية ولغة برمجة هادوب الأساسية هي Java ولكن هذا لا يعني أنه يمكنك كتابة الكود البرمجي فقط في Java بل يمكن البرمجة بلغة C و C++ و Perl و Python و ruby وغيرها، ولكن ستكون لغة جافا أكثر ملاءمةً.

يركز هادوب على نقل الكود البرمجي إلى مكان البيانات بدلاً من النقل المستمر للبيانات أي أصبحت العمليات الحسابية تتم على البيانات الموجودة على نفس الجهاز ولأنَّ نقل البيانات سوف يستغرق وقتاً أطول من إجراء العمليات الحسابية على البيانات[15] ، ومن بين فوائد هادوب يمكننا أن نذكر العناصر التالية[13] :

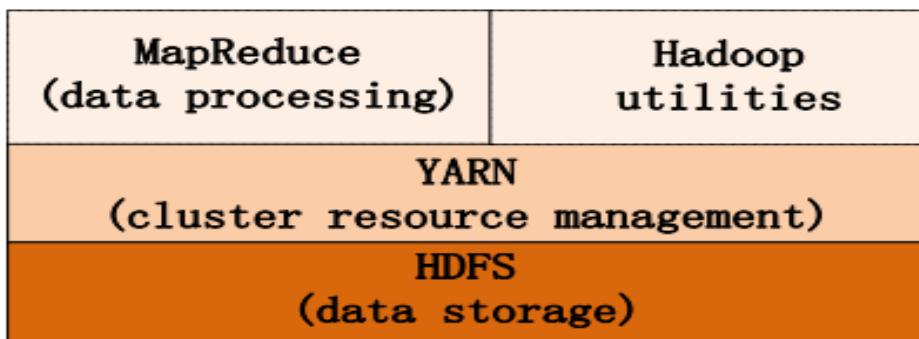
1. **الحجم:** يعمل هادوب على عناقيد كبيرة من الأجهزة الحاسوبية أو خدمات الحوسبة السحابية.
2. **القوة:** يفترض هادوب في تصميمه وجود خلل متكرر في أداء الأجهزة، ويضع السيناريوهات المناسبة للتعامل مع مثل هذه الأضطرابات.
3. **قابلية التوسيع:** هادوب قابل للتتوسيع خطياً لدعم التعامل مع البيانات الضخمة متزايدة الحجم من خلال إضافة المزيد من العقد إلى العنقد.
4. **البساطة:** هادوب يسمح للمستخدمين بكتابة برامجهم المتوازية بسهولة وفعالية.

##### 1-2 مكونات هادوب ([1](Apache Hadoop Components)):

يتكون هادوب من أربعة أجزاء رئيسية:

- **Map-Reduce:** نموذج برمجة يقدم الدعم للمعالجة المتوازية والجدولة المدركة لمكان توضع البيانات والتسامح مع الأعطال وقابلية التوسيع.
- **YARN (Hadoop 2.x):** مدير الموارد الذي يقوم بجدولة المهام وحجز الموارد داخل العنقد.
- **HDFS (Hadoop Distributed File Systems):** نظام ملفات هادوب الموزع الذي يخزن الملفات ويربط كلتها بشكل منطقي.
- **Hadoop Common:** مكتبات جافا الازمة من أجل تشغيل باقي وحدات هادوب.

يعتبر كل من HDFS (التخزين) وما بريديوس (المعالجة) هما المكونان الأساسيان لإطار Apache Hadoop الجانب الأكثر أهمية في هادوب هو أن كلا من HDFS وما بريديوس مصممان مع بعضهما البعض و يتم نشر كل منها بشكل مشترك بحيث يكون هناك كتلة واحدة، وبالتالي توفر القدرة على نقل الحساب إلى البيانات وليس العكس، وبالتالي لا يكون نظام التخزين منفصلاً فعلياً عن نظام المعالجة[2].



الشكل (1-1) مكونات أباثشي هادوب [1].

### 3-1 نظام ملفات هادوب الموزع [3] : HDFS (Hadoop Distributed File System)

نظام (HDFS) هو نظام ملفات قائم على Java يوفر مخزن بيانات قابل للتوسيع وموثوق ومصمم بحيث يغطي عناقيد كبيرة الحجم يوفر وصولاً عالي السرعة إلى البيانات، وهو نظام تخزين لعنقود هادوب فعندما تأتي بياناتٌ جديدةٌ لعنقود يقوم نظام الملفات بتقسيمها إلى أجزاء ويوزع هذه الأجزاء على الخوادم المختلفة المشاركة في العنقود يخزن كل خادم جزءاً صغيراً من مجموعة البيانات الكلية، وينسخ كل جزء من البيانات على أكثر من خادم واحد، وبما أن نظام الملفات هذا يخزن البيانات الكلية بشكلٍ أجزاءٍ صغيرةٍ على مجموعةٍ من الخوادم، فإنَّ مهام التحليل تُوزَّع تفْرِعِياً على كلِّ الخوادم التي تحتوي جزءاً من البيانات الكلية [15].

يُقْرَم كل خادم قيمة جزء البيانات المخزن عليه بشكل متزامن مع بقية الخوادم المشتركة بالبيانات الكلية، ويُعَدِّم النتيجة ليتم تجميعها حتى نحصل على جوابٍ شاملٍ للسؤال المراد طرحه على مجموعة البيانات الكلية، ويتكفل ماب ريديوس بتوزيع العمل وإعادة جمع النتيجة نظام HDFS متسم بدرجة عالية من التسامح مع الفشل وهو مصمم ليتم نشره على أجهزة منخفضة التكلفة. يُنشئ HDFS نسخاً متماثلة متعددة لكل كتلة بيانات ويوزعها على أجهزة الكمبيوتر عبر نظام العنقود لتمكين الوصول الموثوق وال سريع.

### 1-3-1 مزايا نظام ملفات هادوب الموزع [3]:

- ملفات كبيرة جداً: إنَّ هادوب يتعامل مع ملفات تبلغ مئات ميغابايت أو غيغابايت أو تيرابايت في الحجم هناك عناقيد هادوب تعمل اليوم تخزن البئتابايت petabytes من البيانات.
- الوصول المتدايق إلى البيانات بشكل تدفقى Streaming Data Access: يستند HDFS حول فكرة أن نمط معالجة البيانات الأكثر كفاءة هو نمط الكتابة مرة واحدة، والقراءة عدة مرات. يتم عادةً إنشاء مجموعة بيانات أو نسخها من المصدر، ومن ثم يتم إجراء العديد من التحليلات على مجموعة البيانات هذه بمرور الوقت.
- الأجهزة السلعية Commodity hardware: لا تتطلب هادوب أجهزة باهظة الثمن وموثوقة للغاية. إنها مصممة للتشغيل على مجموعات من الأجهزة السلعية (الأجهزة المتأخرة عادةً والتي يمكن الحصول عليها من بائعين متعددين) التي تكون فرصة فشل عقد عبر الكتلة عالية، على الأقل للعناقيد الكبيرة، تم تصميم HDFS لمواصلة العمل دون انقطاع ملحوظ للمستخدم في مواجهة هذا الفشل.
- التأخير المنخفض للوصول إلى البيانات Low-latency data access: التطبيقات التي تتطلب وقتاً قصيراً للوصول إلى البيانات، بعشرات الملي ثانية، لن تعمل بشكل جيد مع HDFS تم تحسين HDFS لتسلیم البيانات بإنتاجية عالية، وهذا قد يكون على حساب التأخير.
- الكتابات المتعددة والتعديلات التعسفية Multiple writers, arbitrary file modifications: يمكن الكتابة والتعديل على الملف في HDFS بواسطة زبون واحد في لحظة زمنية معينة أي لا يوجد دعم لكتاب المتعديين والتعديل المتزامن على الملف وذلك من خلال عقد ايجار بين الزبون والعقدة الرئيسية ويتم دائماً الكتابة في نهاية الملف حيث لا يدعم هادوب تعديلات بإزاحات أو أماكن مختلفة في الملف.
- الكتل Blocks: عند كتابة أي ملف في HDFS ، يتم تقسيمه إلى أجزاء صغيرة من البيانات المعروفة باسم الكتل، يحتوي HDFS على حجم كتلة افتراضي يبلغ 128 ميغابايت والذي يمكن زيارته وفقاً للمطلبات يتم تخزين هذه الكتل في العنقود بطريقة موزعة على العقد المختلفة، ويوفر هذا آلية ماب ريديوس لمعالجة البيانات على التوازي في العنقود [15] .



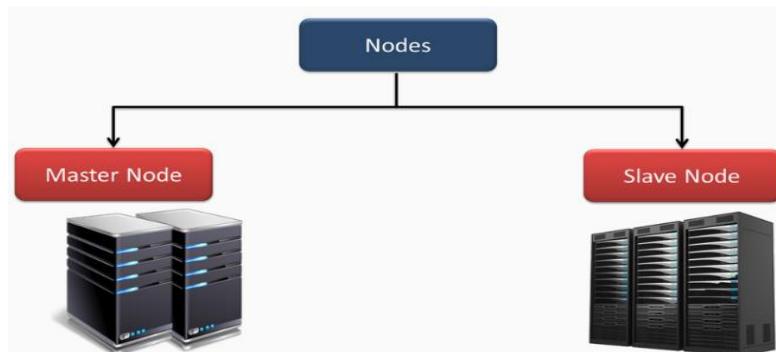
الشكل (2-1) التخزين في نظام ملفات هادوب الموزع [55].

➢ نقل الحساب أرخص من نقل البيانات: يعد الحساب الذي يطلب التطبيق أكثر فاعلية إذا تم تنفيذه بالقرب من البيانات التي يعمل عليها، هذا صحيح بشكل خاص عندما يكون حجم مجموعة البيانات ضخماً يقلل هذا من ازدحام الشبكة ويزيد من معدل النقل الإجمالي للنظام.

➢ التسامح مع الأخطاء fault tolerant: بما أن نظام HDFS يستخدم الكثير من الأجهزة فهذا يمكنه من الاستمرار في العمل حتى ولو تعطل البعض منها [3].

### 3-2 مكونات نظام ملفات هادوب الموزع [55]

يحتوي نظام HDFS على نوعين من العقد يعملان في نمط العامل – والسيد، حيث يتتألف عقد الـ HDFS من NameNode (العقدة الرئيسية) واحدة وعدد من DataNodes (عقد البيانات).



الشكل (4-1) أنواع العقد في نظام ملفات هادوب الموزع [55].

تدير العقدة الرئيسية فضاء أسماء النظام الموزع namespace وتتخزن باستمرار شجرة نظام الملفات والبيانات الوصفية metadata لجميع الملفات والمجلدات على القرص المحلي فيها على شكل ملفين: EditLog وFsImage يسجل في EditLog باستمرار كلّ تغيير يحدث للبيانات الوصفية لنظام الملفات ويُخزن كاملاً في ذلك ربط الكتل بالملفات وخصائص نظام الملفات في ملف يسمى FsImage.

لدى العقدة الرئيسية معرفة أيضاً عن عقد البيانات التي توجد عليها جميع الكتل الخاصة بملف معين وتنظم وصول الزبائن إلى الملفات [15].

تعتبر عقد البيانات DataNodes العاملة في نظام الملفات، حيث يقومون بتخزين الكتل واسترجاعها عندما يتم طلب ذلك (من قبل الزبائن أو العقدة الرئيسية) وتقوم عقدة البيانات أيضاً بإنشاء كتلة وحذفها ونسخها إلى عقد بيانات أخرى بناء على تعليمات من العقدة الرئيسية ويمكن للتطبيق أن يحدد عدد النسخ المتماثلة لملف التي يجب الحفاظ عليها بواسطة HDFS ويسمى عدد النسخ من الملف بعامل النسخ المتماثل replication factor تخزن هذه المعلومات بواسطة العقدة الرئيسية وأيضاً تقوم عقدة البيانات بارسال تقارير إلى العقدة الرئيسية بشكل دوري تحوي قوائم بالكتل التي تقوم بتخزينها.

### 3-3 التخزين المؤقت للكتل [3]Block Caching

عادةً ما تقرأ عقدة البيانات DataNode الكتل من الفرص، ولكن بالنسبة إلى الملفات التي يتم الدخول إليها بشكل متكرر يمكن تخزين الكتل بشكل واضح في ذاكرة التخزين المؤقت لعقدة البيانات (خارج الكومة heap افتراضياً). يتم تخزين نسخة مؤقتة في ذاكرة عقدة بيانات واحدة فقط، على الرغم من أن الرقم قابل للتغير على أساس كل ملف يمكن لمجدول الأعمال (لـ ماب ريديوس و سبارك والأطر الأخرى) الاستفادة من الكتل المخزنة مؤقتاً عن طريق تشغيل المهام على عقدة البيانات التي تم تخزين الكتلة عليها مؤقتاً وذلك لزيادة أداء القراءة.

### 1-3-4 الوضع الآمن [15]:Safemode

عند بدء التشغيل تدخل العقدة الرئيسية حالة خاصة تسمى الوضع الآمن، لا يحدث نسخ كتل البيانات عندما تكون العقدة الرئيسية في الوضع الآمن. تستقبل العقدة الرئيسية رسائل نبضة القلب Heartbeat ورسائل تقارير الكتل Blockreport من عقد البيانات تحتوي رسالة تقرير الكتل على قائمة كتل البيانات التي تستضيفها عقد البيانات، وتملك كل كتلة حد أدنى محدد من عدد النسخ، عندما يتم الوصول إلى شرط الحد الأدنى من عدد النسخ للكتل + 30 ثانية إضافية تخرج العقدة الرئيسية من الوضع الآمن، ثم يحدد قائمة كتل البيانات التي لا تزال أقل من العدد المحدد من النسخ المتماثلة وينسخها إلى عقد بيانات أخرى.

### 1-3-5 بيانات نظام الملفات الوصفية [55]:Metadata

يستخدم HDFS سجل يسمى سجل التحرير EditLog وذلك حتى يسجل باستمرار كل تغيير يحدث على البيانات الوصفية metadata لنظام الملفات، مثلًا إنشاء ملف جديد في HDFS يجعل العقدة الرئيسية تدرج سجلاً في EditLog يشير إلى ذلك، وكذلك تغيير عامل النسخ يسبب إدراج سجل جديد في EditLog. يُخزن كامل فضاء الأسماء namespace بما في ذلك ربط الكتل بالملفات وخصائص نظام الملفات في ملف يسمى FsImage يُخزن كل من EditLog و FsImage ملفات في نظام الملفات المحلي للعقدة الرئيسية وعند بدء تشغيلها تقوم بتحميل فضاء الأسماء namespace من آخر FsImage محفوظة في ذاكرتها، وتطبق وتدمج على EditLog لتعطي namespace جديد، ثم تدخل الوضع الآمن.

## 4. مقارنة 1 مع YARN [3]

يشار في بعض الأحيان إلى التنفيذ الموزع لماب ريديوس في النسخة الأصلية من هادوب (الإصدار 1 وما قبله) باسم "MapReduce 1" التمييز عن MapReduce 2 وهو التمثيل الذي يستخدم YARN في 2 Hadoop وما بعده ، هناك نوعان من الوكلاء في MapReduce 1 وهما Jobtracker و Tasktrackers متعدد العمل الذي يتحكم في عملية تنفيذ المهمة وواحد أو أكثر من متعقبات المهام TaskTrackers ينسق متعقب العمل كافة المهام التي يتم تشغيلها على النظام عن طريق جدولة المهام لتشغيلها على متعقبات المهام، تقوم متعقبات المهام TaskTrackers بتنفيذ المهام وإرسال تقارير التقدم إلى متعقب العمل، والذي يحتفظ بسجل للتقدم العام لكل عمل، إذا فشلت إحدى المهام فيمكن لمتعقب العمل أن يعيد جدولة المهام على متعقب مهام آخر، في MapReduce 1 يعتني متعقب العمل بجدولة المهام (اسناد المهام إلى متعقبات المهام) ومراقبة تقدم المهام (تنبيه المهام، وإعادة تشغيل المهام الفاشلة أو البطيئة). على النقيض من ذلك، في YARN يتم التعامل مع هذه المسؤوليات بواسطة كيانات منفصلة: مدير الموارد ومدير التطبيق (واحد لكل عمل ماب ريديوس) إن متعقب العمل هو المسؤول أيضاً عن تخزين تاريخ الأعمال المكتملة، على الرغم من أنه من الممكن تشغيل مخدم لحفظ تاريخ الأعمال ككيان منفصل لتخفيض الحمل عن متعقب الأعمال. في YARN هناك خادم يدعى مخدم المخطط الزمني يقوم ب تخزين محفوظات التطبيق .

الجدول (1-1) مقارنة 1 مع YARN .[3] MapReduce1

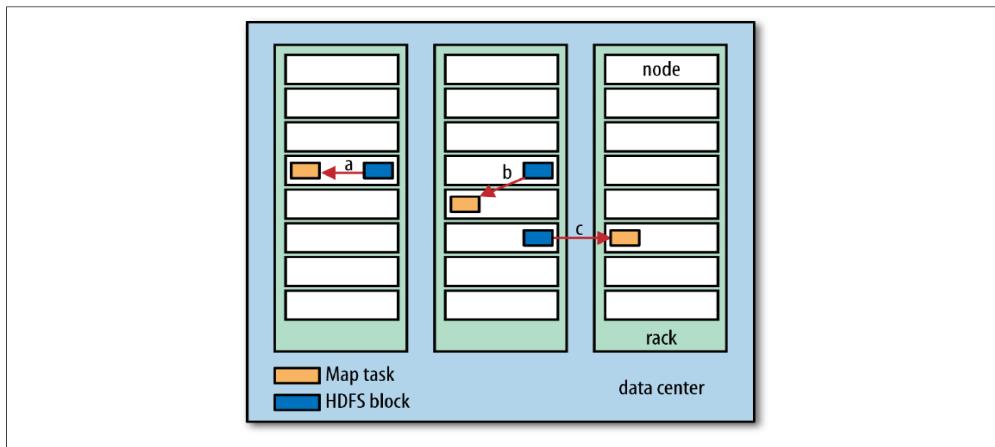
MapReduce 1	YARN
Jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
Slot	Container

- تم تصميم YARN لحل العديد من القيود في 1 MapReduce تتضمن فوائد استخدام YARN ما يلي:
  - قابلية التوسيع Scalability : يمكن أن تعمل YARN على عناقيد أكبر من 1 MapReduce تبلغ 1 اختلافات قابلية التوسيع عند الوصول إلى 4000 عقدة و 40000 مهمة تتبع من حقيقة أنه على متعد العمل أن يدير كل من المهام والأعمال، تتغلب YARN على هذه القيود بحكم تصميمها التقسيمي لمدير الموارد ومدير التطبيق، وهي مصممة لتصل إلى 10000 عقدة و 100000 مهمة، وعلى النقيض من متعد العمل فإن كل نسخة من تطبيق (عمل ماب ريديوس) لديه مدير تطبيق مخصص لها والذي يعمل طوال مدة التطبيق .
  - التوافر Availability : عادةً ما يتتحقق التوفير العالي (HA) عن طريق نسخ أو تكرار الحالة المطلوبة إلى وكيل آخر لتولي العمل المطلوب لتوفير الخدمة، في حالة فشل الوكيل الأول، ومع ذلك فإن السرعة الكبيرة في تغيير حالة ذاكرة متعد العمل (كل حالة مهمة يتم تحديثها كل بضع ثوان) يجعل من الصعب للغاية تحقيق الإتاحة في متعد الأعمال، مع تقسيم المسؤوليات الوظيفية بين مدير الموارد ومدير التطبيق في YARN ، أصبحت الخدمة متاحة بشكل كبير وذلك عن طريق اتباع مبدأ فرق تسد أي أن مسألة الإتاحة أصبحت تتعلق بإتاحة مدير الموارد وإتاحة مدير التطبيق [17] .
  - الاستخدام Utilization: في MapReduce1 يتم إعداد كل متعد كل مهام بتخصيص ثابت لـ "فتحات" ذات حجم ثابت، والتي يتم تقسيمها إلى فتحات لمهام المقابلة وفتحات لمهام الاختزال في وقت التهيئة . لا يمكن استخدام فتحة المقابلة إلا لتشغيل مهمة مقابلة، ولا يمكن استخدام فتحات الاختزال إلا ل مهمة الاختزال. في YARN يدير مدير العقدة مجموعة من الموارد، بدلاً من عدد محدد من الفتحات. ماب ريديوس الذي يعمل على YARN لن يصل إلى الحالة التي يجب أن تنتظر فيها مهمة الاختزال نظراً لأن فتحات المقابلة هي فقط المتاحة في العنقود كما يحدث في MapReduce1 فإذا كانت الموارد اللازمة لتشغيل المهمة متاحة، فإن التطبيق سيكون مقبولاً لتشغيلهعلاوةً على ذلك، فإن الموارد في YARN من النوع fine grained لذا يمكن للتطبيق تقديم طلب بما يحتاج إليه، بدلاً من وجود فتحة غير قابلة للتجزئة، والتي قد تكون كبيرة جداً (ما قد يتسبب في فشل) للمهمة المحددة.
  - متعدد الإيجار Multitenancy : في بعض التواحي، فإن أكبر فائدة من YARN هو أنه يفتح هادوب إلى أنواع أخرى من التطبيقات الموزعة خارج ماب ريديوس، إذ يعتبر ماب ريديوس واحد فقط من تطبيقات YARN العديدة، من الممكن أيضاً للمستخدمين تشغيل إصدارات مختلفة من ماب ريديوس على نفس عنقود YARN مما يجعل عملية ترقية ماب ريديوس أكثر قابلية للإدارة.

: [3][15] Data Flow (البيانات) 5.1

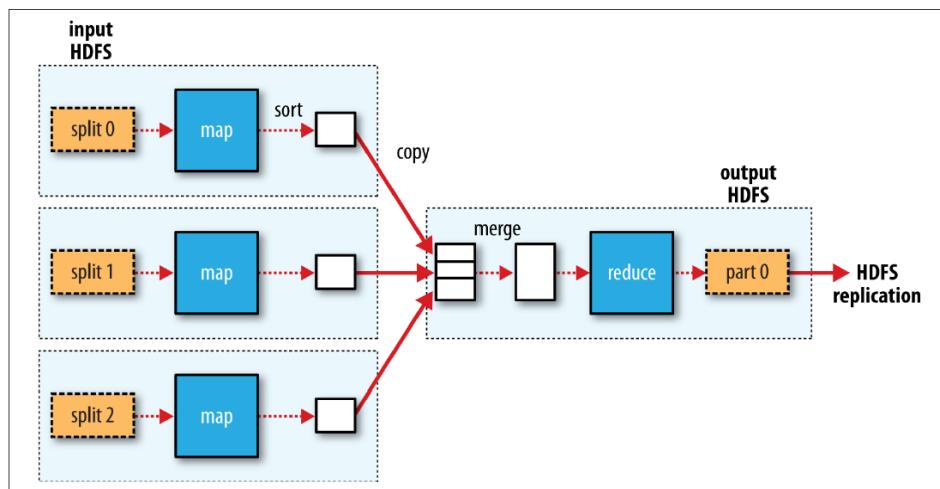
يدير هادوب العمل ب التقسيمه إلى مهام وهناك نوعان من المهام هما مهام المقابلة و مهام الاختزال و تتم جدولة المهام باستخدام YARN و تشغيلها على العقد في العنقد فإذا فشلت إحدى المهام، فسيتم إعادة جدولتها تلقائياً لتعمل على عقدة مختلفة يقسم هادوب دخل عمل ماب Ridiyos إلى أجزاء ذات حجم ثابت تسمى أجزاء الدخل input splits يقوم هادوب بإنشاء مهمة مقابلة و احدة لكل حزء من الدخل، والتى تقوم بتشغيلتابع المقابلة المعرف من قبل المستخدم لكل سجل.

يبذل هاوب قصارى جهده لتشغيل مهمة المقابلة على عقدة التي تتوارد عليها بيانات الإدخال HDFS لأنها لا تستخدم عرض نطاق ذي القيمة، وهذا ما يسمى data locality optimization ومع ذلك في بعض الأحيان تعمل جميع العقد التي تستضيف نسخاً متماثلة من كتلة HDFS لدخل مهمة المقابلة على تشغيل مهام مقابلة أخرى، لذا سيبحث مجدول المهام عن فتحة مقابلة فارغة على عقدة في نفس الرف وفي بعض الأحيان حتى هذا غير ممكن لذلك يتم استخدام عقدة off-rack خارج الرف مما يؤدي إلى نقل البيانات بين الرفوف الاحتمالات الثلاثة موضحة في الشكل التالي:



الشكل (5-1) تدفق البيانات في [3] HDFS

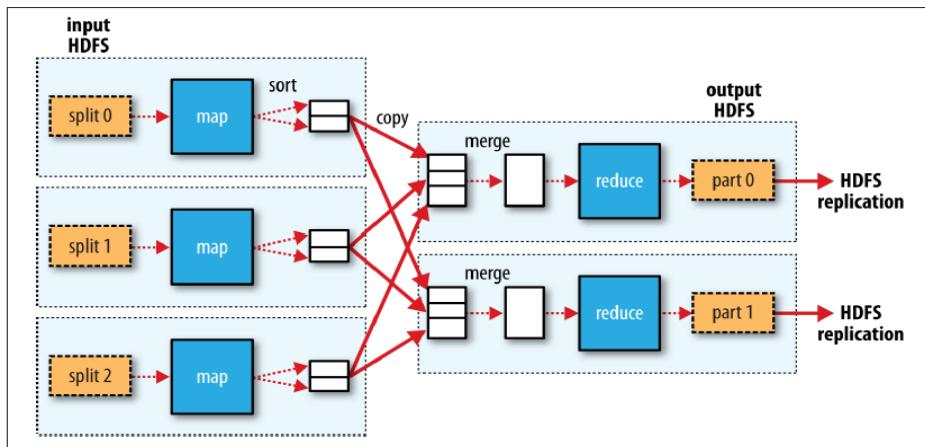
لا تمتلك مهام الاختزال ميزة محلية البيانات (data locality) عادةً ما يكون الدخول إلى مهمة الاختزال واحدة هو خرج جميع مهام المقابلة. في المثال الحالي لدينا مهمة اختزال واحدة يتم تعذيتها بواسطة جميع مهام المقابلة، لذلك يجب نقل مخرجات المقابلة التي تم فرزها عبر الشبكة إلى العقدة إلى العقدة حيث يتم تشغيل مهمة الاختزال عليها، حيث يتم دمج النتائج ثم تمريرها إلى تابع الاختزال المعرف من قبل المستخدم يتم تخزين ناتج الاختزال عادةً في HDFS من أجل الموثوقية حيث من أجل كل كتلة من خرج تابع الاختزال يتم تخزين النسخة المتماثلة الأولى لها على العقدة المحلية، مع تخزين النسخ المتماثلة الأخرى على العقد خارج الرف (off-rack nodes) لزيادة الاعتمادية، وبالتالي فإن كتابة خرج تابع الاختزال يستهلك عرض النطاق الترددي للشبكة، ولكن فقط بقدر استهلاكه عند كتابة كتلة جديدة على خط أنابيب HDFS العادي.



الشكل (6-1) تدفق البيانات من عدة مهام مقابلة إلى مهمة اختزال وحيدة[3]

عندما يكون هناك مختزلات متعددة، تقوم مهام المقابلة بتقسيم مخرجاتها، كل منها ينشئ قسماً واحداً لكل مهمة اختزال . يمكن أن يكون هناك العديد من المفاتيح (والقيمة المرتبطة بها) في كلّ قسم، ولكن سجلات أي مفتاح تتواجد ضمن قسم واحد يمكن التحكم في التقسيم بواسطة تابع التقسيم المعرف من قبل المستخدم ولكن عادةً يعمل المقسم الأفتراضي.

يوضح الشكل التالي تدفق البيانات في حالة وجود مهام اختزال متعددة حيث يوضح هذا الرسم البياني السبب أن تسمية تدفق البيانات بين مهام المقابلة والاختزال بالخلط "shuffle" حيث أن كل مهمة اختزال تغذيها العديد من مهام المقابلة.



الشكل (7-1) تدفق البيانات من عدة مهام مقابلة إلى عدة مهام اختزال [3].

**6-1تابع التجميع Combiner Functions:** العديد من أعمال ماب ريديوس تكون محدودةً بعرض النطاق التردي المتاح على العنقود، لذلك من أجل تقليل البيانات المنقولة بين مهام المقابلة والاختزال يسمح هادوب للمستخدم بتعريف تابع تجميع ليتم تشغيله على مخرجات مهمة المقابلة ويشكل خرج تابع التجميع دخل تابع الاختزال.

#### 6-1-1 كيفية عمل ماب ريديوس في هادوب :MapReduce in Hadoop

1- يقوم الزبون بتشغيل عمل ماب ريديوس وذلك عن طريق الخطوات التالية:

- يطلب الزبون من مدير الموارد معرف تطبيق جديد.
- يتم التحقق من الدليل الذي سيحفظ به الخرج، فمثلاً إذا كان موجوداً بالفعل، فلن يتم إرسال المهمة ويظهر خطأ في البرنامج.
- يتم تجزئة دخل العمل إلى أجزاء .
- ثم يتم نسخ الموارد الازمة لتشغيل العمل كملف JAR الخاص بالعمل وملف الإعداد وأجزاء ملف الدخل إلى نظام الملفات الموزع (HDFS) في دليل يسمى بعد تحديد معرف العمل.
- ثم يتم استدعاء العمل.

#### 2- يقوم مدير الموارد YARN بعملية تخصيص موارد الحوسبة على العنقود [3] :

عندما يتم استدعاء التطبيق فإنه يسلم الطلب إلى مجدول YARN الذي يقوم بتخصيص حاوية للعمل وبعدها يقوم مدير الموارد بإطلاق مدير التطبيق ليعمل تحت إدارة مدير العقدة.

إن مدير التطبيق هو تطبيق جافا الذي يقوم بتهيئة العمل وإنشاء مجموعة من الأغراض التي تتبع تقديم العمل حيث يتلقى مدير التطبيق تقارير عن تقديم وانتهاء المهام، ثم يسترد أجزاء دخل العمل من نظام الملفات الموزع وينشئ مهمة مقابلة من أجل كل جزء أما عدد مهام الاختزال يحدد بالخاصية mapreduce.job.reduces .

يقرر مدير التطبيق كيفية تشغيل المهام التي يتكون منها العمل فإذا كان العمل صغير الحجم فقد يختار تشغيله في نفس JVM يحدث هذا عندما يرى مدير التطبيق أن النفقات العامة لتخصيص المهام وتشغيلها في حاويات جديدة تفوق المكاسب التي يمكن تحقيقها في تشغيلها بالتوازي مقارنة بتشغيلها بشكل تسلسلي على عقدة واحدة.

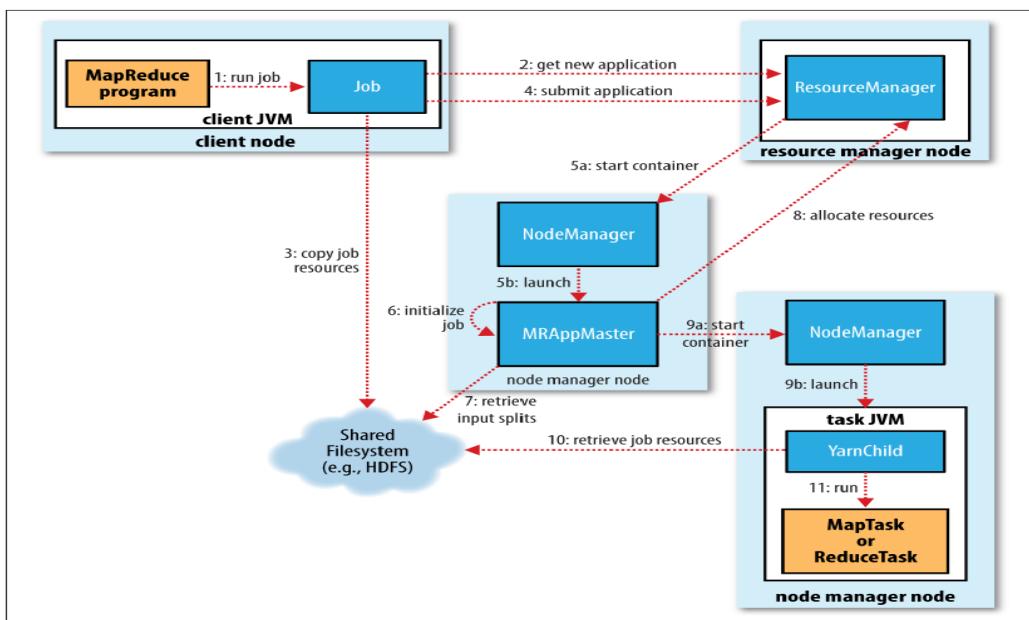
3- يقوم مدير تطبيق ماب ريديوس بتنسيق تشغيل المهام حيث يتم تشغيل مدير التطبيق ومهام ماب ريديوس في حاويات يتم جدولتها بواسطة مدير الموارد ويتم إدارتها بواسطة مدير العقدة:

عندما يحتاج العمل إلى موارد إضافية يقوم مدير التطبيق بطلب حاويات جديدة من مدير الموارد من أجل مهام المقابلة والاختزال.

يتم إجراء طلبات مهام المقابلة أولاً وبأولوية أعلى من تلك الخاصة بمهام الاختزال نظراً لأنه يجب إكمال جميع مهام المقابلة قبل بدء مرحلة الاختزال .

#### 4- تنفيذ المهمة:

- بمجرد تعيين موارد حاوية على عقدة معينة لمهمة ما بواسطة مدير الموارد. يبدأ مدير التطبيق الحاوية عن طريق الاتصال بمدير العقدة. يتم تنفيذ المهمة بواسطة تطبيق Java الذي يكون صفة الرئيسي هو YarnChild ثم يتم تشغيل مهام المقابلة أو الاختزال (الخطوة 11).
  - يعمل YarnChild في JVM مخصص بحيث لا يؤثر أي خلل في توابع المقابلة والاختزال المعرفة من قبل المستخدم على مدير العقدة عن طريق التسبب في تعطله أو تعليقه على سبيل المثال.
  - يمكن لكل مهمة تنفيذ إجراءات التهيئة (setup) والتسلیم (commit) والتي يتم تشغيلها في نفس JVM بالنسبة إلى المهام المستندة إلى الملفات، ينقل إجراء التسلیم مخرجات المهمة من موقعها المؤقت إلى موقعها النهائي يضمن برتوکول التسلیم أنه عند تمكّن تنفيذ المضاربة speculative execution يتم تسلیم خرج واحدة فقط من المهام المكررة ويتم إحباط الأخرى.
- 5- اكتمال العمل : Job Completion**
- عندما يتلقى مدير التطبيق إشعاراً بانتهاء المهمة الأخيرة للعمل، فإنه يغير حالة المهمة إلى "ناجحة"، ويتم طباعة رسالة تخبر المستخدم بانتهاء العمل بنجاح وتم طباعة إحصائيات العمل والعدادات.
  - وأخيراً يقوم مدير التطبيق وحاويات المهام عند إكمال العمل بتنظيف حالة العمل الخاصة بهم (بحيث يتم حذف المخرجات المرحلية)، ويتم استدعاء الدالة commitJob() و يتم أرشفة معلومات العمل من خلال خادم تاريخ المهام (لتمكين المستخدمين من الوصول إليهم فيما بعد).



الشكل (8-1) كيفية تشغيل هادوب لعمل ماب ريديوس[3].



## الفصل الثاني

### التسامح مع الأعطال في هادوب

### Hadoop Fault Tolerance

#### 2-1 مقدمة:

يُعرف مفهوم التسامح مع الفشل بأنه قدرة النظام على الاستمرار في العمل بشكل صحيح دون فقدان أي بيانات حتى إذا فشلت بعض مكونات النظام في الأداء بشكل صحيح، ومن الصعب جدًا تحقيق التسامح بنسبة 100٪ ولكن يمكن التغاضي عن الأعطال إلى حد ما. صمم كل من ماب ريديوس وHDFS لمتابعة العمل في حالات مواجهة أي فشل في النظام، حيث يقوم هادوب بشكل مستمر بمراقبة البيانات المخزنة في العنقود. في حال أصبح أي من الخوادم غير متاح، أو فشل محرك الأقراص، أو تألفت البيانات نتيجةً مشاكل برمجية أو عتادية، يقوم نظام ملفات HDFS تلقائيًا باستعادة البيانات نفسها من أحد الخوادم الأخرى التي حُرِّزَت عليها نسخة احتياطية عند عملية التقسيم، وبالتالي عندما تكون عملية تحليل قيد العمل يقوم ماب ريديوس بمراقبة التقدم على كل الخوادم المشاركة بالعملية، وفي حال كان أحد هذه الخوادم بطريقًا في إعادة النتيجة أو فشل في إكمال مهمته، يقوم ماب ريديوس فورًا بتوجيه خادم آخر حُرِّزَ عليه نفس الجزء من البيانات المخزنة على الأول لبدء العمل بدلاً عنه، وبالتالي ونظرًا للطريقة التي يعمل بها HDFS و ماب ريديوس ، تقدم شركة هادوب خوادم موثوقة، معالجة للفشل وقابلة للتطوير لتخزين وتحليل البيانات بكلفة منخفضة جدًا.

#### 2-2 التسامح مع الفشل في نظام ملفات هادوب الموزع :HDFS Fault Tolerance

إن HDFS متسامحة للغاية مع الأعطال وتم تصميمها بحيث يتم نشرها على أجهزة منخفضة التكلفة وبالتالي فشل الأجهزة هو القاعدة وليس الاستثناء. قد يحتوي HDFS على مئات أوآلاف من الأجهزة، كل منها يخزن جزءًا من بيانات نظام الملفات، وإن حقيقة وجود عدد هائل من المكونات وأن كل مكون له احتمال فشل غير قابل للتجاهل يعني أن بعض مكونات HDFS قد تكون غير فعالة [1] ، ولذلك فإن اكتشاف الأعطال والاسترداد التلقائي السريع منها هو هدف أساسي لـ HDFS كذلك يعتبر الغرض الرئيسي من النظام هو إزالة حالات الفشل الشائعة، والتي تحدث بشكل متكرر وتوقف النظام عن العمل، وإن أهم مزايا استخدام هادوب هي وجود اثنين من الطرق الرئيسية التي تستخدم لإنتاج التسامح مع الفشل فيه وهي تكرار البيانات ونقطة التقسيم [55] الأنواع الثلاثة من الإخفاقات في HDFS هي:

- فشل العقدة الرئيسية .NameNode
- فشل عقدة البيانات .DataNode
- فشل أنواع الشبكة.

#### 2-2-1 فشل العقدة الرئيسية والتسامح معه :[3][53] NameNode Failure and Fault Tolerance

بدون العقدة الرئيسية NameNode لا يمكن استخدام نظام الملفات فإذا تم القضاء على الجهاز الذي يشغل NameNode سيتم فقد جميع الملفات الموجودة على نظام الملفات نظرًا لعدم وجود طريقة لمعرفة كيفية إعادة بناء الملفات من الكتل المتواجدة على DataNodes .

فالعقدة الرئيسية تمثل نقطة واحدة من الفشل (SPOF) single point of failure إذا فشلت فإن جميع الزبائن بما في ذلك أعمال ماب ريديوس لن يتمكنوا من قراءة الملفات أو كتابتها أو سردها لأن NameNode هو المستودع الوحديد للبيانات الوصفية والرابط بين كل ملف وكتلاته، في مثل هذا الحدث سيكون نظام هادوب بأكمله خارج الخدمة حتى إحضار جديدة لهذا السبب من المهم جعل العقدة الرئيسية مرنة للفشل حيث يوفر هادوب آليتين لذلك ويتم تحقيق NameNode التسامح مع فشل العقدة الرئيسية في HDFS من خلال:

1. العقدة الرئيسية الثانوية (Secondary Name Node[Hadoop1]): الطريقة الأولى هي إجراء نسخ احتياطي للملفات التي تشكل الحالة المستمرة (file system metadata) للبيانات الوصفية لنظام الملفات. يمكن إعداد

هادوب بحيث تكتب العقدة الرئيسية البيانات الوصفية إلى أنظمة ملفات متعددة الإعداد الذي يتم اختياره عادةً هو الكتابة على القرص المحلي وكذلك على نظام NFS بعيد، و من الممكن أيضاً تشغيل عقدة رئيسية ثانوية Secondary NameNode، على الرغم من اسمها إلا أنها لا تعمل كعقدة رئيسية بل يتمثل دورها الرئيسي في دمج صورة فضاء الأسماء (namespace image) مع سجل التعديل (EditLog) بشكل دوري لمنع سجل التعديل من أن يصبح كبيراً عادةً ما يتم تشغيل العقدة الرئيسية الثانوية على جهاز فيزيائي منفصل لأنه يتطلب نفس متطلبات العقدة الرئيسية من وحدة المعالجة المركزية والذاكرة لتنفيذ الدمج ويحتفظ بنسخة من صورة فضاء الأسماء التي تم دمج التغييرات معها، والتي يمكن استخدامها في حالة فشل العقدة الرئيسية، ومع ذلك فإن المعلومات التي تحويها العقدة الثانوية تعتبر غير كاملة لذلك عند فشل العقدة الرئيسية سوف يتم خسارة فقط التغييرات الحاصلة منذ آخر نقطة استعادة check point قامت به العقدة الرئيسية الثانوية، لن تتمكن العقدة الرئيسية الجديدة من تخديم الطلبات إلى أن يقوم (i) بتحميل صورة فضاء الأسماء الخاصة به في الذاكرة، (ii) أعاد تطبيق سجل التعديل الخاص به، و (iii) تلقيه reports كافية من العقد الرئيسية لترك الوضع الآمن. يمكن أن يأخذ الوقت الذي يستغرقه بدء تشغيل العقدة الرئيسية في العناقيد الكبيرة التي تحتوي على العديد من الملفات والقتل حوالي 30 دقيقة أو أكثر مما يسبب مشكلة كبيرة في توافر نظام ملفات هادوب الموزع.

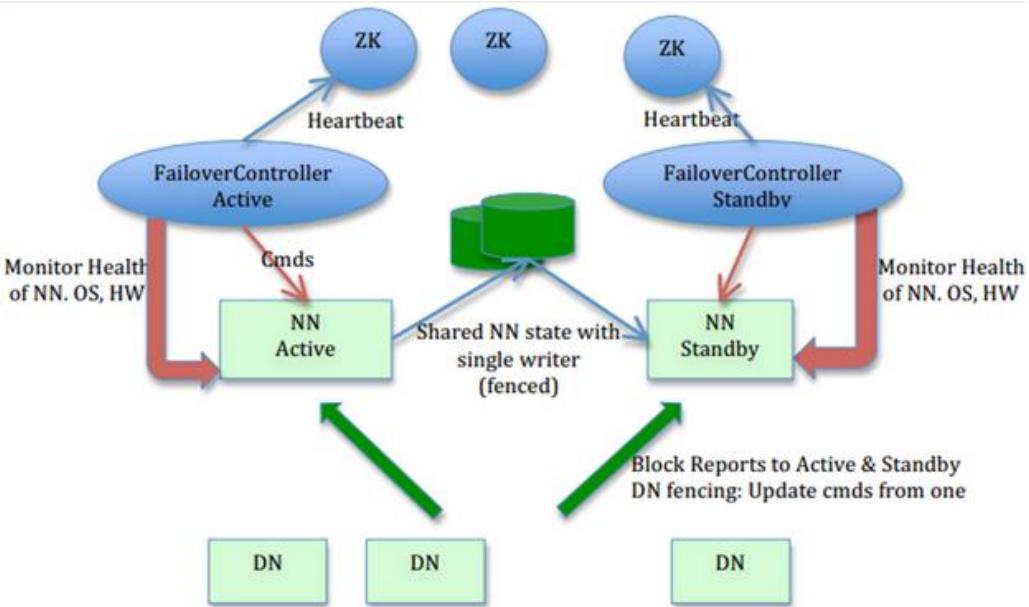
**2. عقدة الاستعداد الرئيسية (Standby Name Node Hadoop2) [39]**: إن الجمع بين نسخ البيانات الوصفية الموجودة على العقدة الرئيسية على أنظمة ملفات متعددة واستخدام عقدة رئيسية ثانوية لإنشاء نقاط تقنيات يحمي من فقد البيانات، ولكنه لا يعطي توفرًا كبيرًا لنظام الملفات [10] تعامل 2 مع هذا الموقف عن طريق تقديم دعم للتوافر العالي لـ HDFS وذلك عن طريق وضع زوج من العقد الرئيسية في وضع active-standby، وفي حالة فشل العقدة الرئيسية النشطة تتحمل العقدة الـ standby واجباتها لمواصلة خدمة طلبات الزبائن دون انقطاع كبير ويوجد هناك بعض التغييرات في البنية التحتية اللازمة لحدث ذلك وهي [40] :

► يجب أن تستخدم العقدتان الرئيسيةان تخزنَا مشاركةً عالي التوفير لمشاركة سجل التعديل EditLog عندما يتم تفريذ أي تعديل في فضاء الأسماء بواسطة العقدة النشطة، فإنه يضيف سجل جديد بالتعديل إلى ملف سجل التحرير المخزن في الدليل المشترك تشاهد عقدة الاستعداد standby باستمرار هذا الدليل لإجراء التعديلات وعندما ترى التعديلات فإنها تطبقها على فضاء الأسماء الخاص بها في حالة تجاوز الفشل failover تضمن عقدة الاستعداد قراءة جميع التعديلات من وحدة التخزين المشتركة قبل ترقيتها إلى الحالة النشطة ويفصل ذلك مزامنة حالة فضاء الأسماء بشكل كامل قبل حدوث عملية تجاوز الفشل.

► يجب أن ترسل عقد البيانات تقارير الكتل block إلى كل من العقدة الرئيسية وعقدة الاستعداد نظرًا لأنه يتم تخزين الروابط بين الكتل block mappings في ذاكرة العقدة الرئيسية وليس على القرص.  
► يجب إعداد الزبائن للتعامل مع الفشل في العقدة الرئيسية باستخدام آلية شفافة للمستخدمين.

### 2-1-2-1 تجاوز الفشل والمبارزة :[3] Failover and fencing

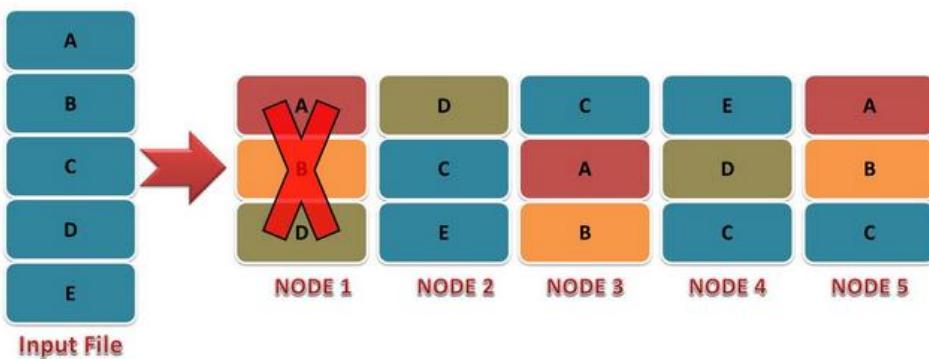
تتم إدارة الانتقال من العقدة الرئيسية النشطة إلى عقدة الاستعداد بواسطة كيان جديد في النظام يسمى متحكم تجاوز الفشل (failover controller) هناك العديد من وحدات التحكم بتجاوز الفشل ولكن افتراضياً يستخدم هادوب ZooKeeper للتتأكد من أنّ هناك عقدة رئيسية واحدة نشطة تعمل في وقت واحد. كل عقدة رئيسية تشغيل متحكم بعملية تجاوز الفشل وحيد مهمته مراقبتها (باستخدام آلية ضربات قلب بسيطة) وإطلاق failover عندما تفشل يمكن أيضًا بدء الفشل يدوياً بواسطة المسؤول، على سبيل المثال في حالة الصيانة الروتينية ويدعى هذا النوع من الفشل بالفشل المرغوب فيه. في حالة حدوث فشل غير مرغوب فيه من المستحيل التأكد من أنّ سبب الفشل هو ايقاف تشغيل العقدة الرئيسية النشطة، على سبيل المثال يمكن أن تؤدي الشبكة البطيئة إلى تشغيل عملية الانتقال لتجاوز الفشل failover [42] على الرغم من استمرار تشغيل العقدة الرئيسية النشطة والتي تظن أنها لا تزال هي العقدة الرئيسية النشطة. إنّ عملية ضمان أنّ العقدة الرئيسية النشطة سابقاً لن تلحق أي ضرر أو تسبب فساد في النظام تدعى باسم المبارزة (fencing) يسمح QJM فقط لعقدة رئيسية واحدة بالكتابة إلى سجل التحرير في وقت واحد، ومع ذلك لا يزال من الممكن للعقدة الرئيسية النشطة سابقاً تقديم طلبات قراءة تالفة للزبائن أو العملاء، لذا فإن إعداد أمر SSH fencing الذي سيؤدي إلى قتل العقدة الرئيسية يعد فكرة جيدة كما يمكن إلغاء وصول العقدة النشطة سابقاً إلى نظام الملفات المشترك عن طريق تعطيل منفذ الشبكة الخاص بها بواسطة الأوامر عن بعد.



الشكل (2-1) تجاوز الفشل والمبارزة في HDFS [3]

## 2-2-2 فشل عقدة البيانات والتسامح معه [49][55]:DataNode Failure and Fault Tolerance

ترسل كل عقدة بيانات رسالة نبضة قلب Heartbeat إلى العقدة الرئيسية بشكل دوري (كل 3 ثواني) والتي تعني أنها تعمل بشكل صحيح، يمكن أن يسبب فشل الشبكة أو فشل عقدة البيانات إلى عدم تلقى العقدة الرئيسية رسالة Heartbeat إذا لم تتلقي العقدة الرئيسية رسالة نبض قلب لمدة عشر دقائق تعتبر عقدة البيانات ميتة وبياناتها غير متوفرة لـ HDFS ولا تعيد العقدة الرئيسية توجيه أي طلبات IO جديدة لها، قد يسبب موت عقدة البيانات إلى هبوط عامل النسخ لبعض الكتل إلى أقل من القيمة المحددة، تتبع العقدة الرئيسية باستمرار أي كتل تحتاج إلى نسخ متماثل وتبدأ بالنسخ كلما لزم الأمر، قد تنشأ ضرورة إعادة النسخ المتماثل بسبب العديد من الأسباب: قد تصبح عقدة البيانات غير متوفرة، قد تختلف النسخة المتماثلة، وقد يفشل القرص الصلب لعقدة البيانات، أو قد يزيد عامل النسخ المتماثل لملف.



الشكل (2-2) فشل عقدة بيانات في نظام ملفات هادوب الموزع [49].

## 2-2-2-1 تكرار كتل البيانات [55][15] Data Block Replication

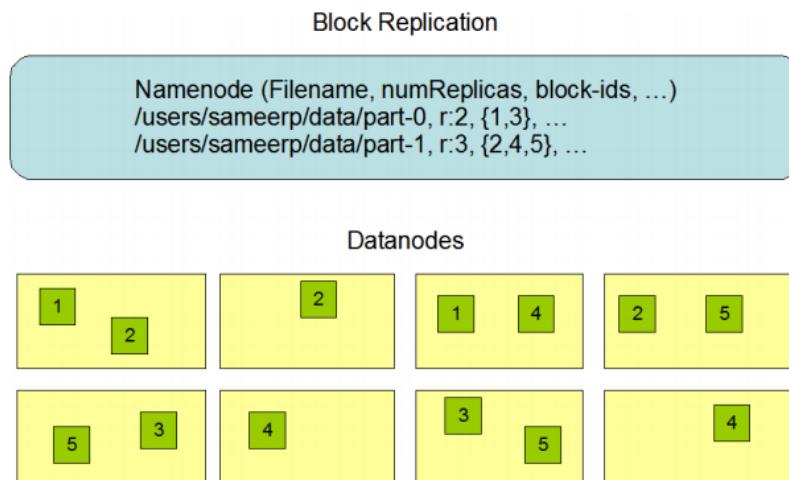
تم تصميم HDFS لتخزين الملفات الكبيرة جداً عبر الأجهزة في عنقود كبير وبشكل موثوق به، يقوم بتخزين كل ملف كتسلسل كتل في العنود لها نفس الحجم ما عدا آخر كتلة في الملف، يتم نسخ كتل الملف من أجل تحقيق التسامح مع الفشل والموثوقية والتوافر العالي، يمكن تحديد حجم الكتلة وعامل النسخ المتماثل لكل ملف، يمكن للتطبيق تحديد عدد النسخ المتماثلة لملف، كما يمكن تحديد عامل النسخ المتماثل في وقت إنشاء الملف ويمكن تغييره فيما بعد . وبشكل

افتراضي، يكون عامل النسخ المتماثل لـ HDFS هو 3. تقوم العقدة الرئيسية باتخاذ جميع القرارات بشأن تكرار الكتل وكما أنه يتلقى بشكل دوري رسائل نبضات قلب وتقارير بالكتل Block report من كل من عقد البيانات في العنقود. تلقي نبضات يدل على عمل عقدة البيانات بشكل صحيح كمل تحتوي Block report على قائمة بكافة الكتل على عقدة البيانات [2]. إن تكرار البيانات يوفر الاسترداد الفوري من الفشل ولكن لتحقيق مثل هذا النوع من التسامح يتم استهلاك كمية كبيرة من الذاكرة في تخزين البيانات على العقد المختلفة أي إهدار كمية كبيرة من الذاكرة والموارد، ونظرًا لتكرار البيانات عبر العقد المختلفة، فقد يكون هناك احتمال لعدم اتساق البيانات، ولكن لأن هذه التقنية توفر انتعاشًا فوريًا وسريعًا من حالات الفشل، لذلك تستخدم بشكل متكرر مقارنةً بنقاط التفتيش [55]. من المحتمل أن تصل كتلة البيانات التي تم جلبها من

DataNode تالفة بسبب:

- أخطاء في جهاز التخزين
- أخطاء الشبكة
- برمج buggy

عندما ينشئ الزبون ملف HDFS يحسب Checksum لكل كتلة ويخزنها في ملف مخفي منفصل في نفس namespace. عندما يسترد الزبون محتويات الملف يتحقق من أن البيانات التي استقبلها من كل DataNode تطابق المخزن إذا لم يكن كذلك يمكن للزبون اختيار استرداد هذه الكتلة من آخر يحوي على نسخة من تلك الكتلة.



الشكل (2-3) تكرار كتل البيانات عبر عنقود الـ HDFS [45]

تحوي عناقيد HDFS الكبيرة على مجموعة من أجهزة الكمبيوتر التي تنتشر بشكل شائع عبر العديد من الرفوف ويجب أن يمر الاتصال بين عقدتين في رفوف مختلفة عبر المبدلات الشبكية Switches. في معظم الحالات يكون عرض النطاق الترددية للشبكة بين الأجهزة الموجودة في نفس الرف أكبر من عرض نطاق الشبكة بين الأجهزة في الرفوف المختلفة. تحدد العقدة الرئيسية معرف الرف الذي توجد عليه كل عقدة بيانات. إن سياسة وضع النسخ المتماثلة على رفوف فريدة بسيطة ولكنها ليست مثالية فهي تمنع خسارة البيانات عند فشل كامل الرف وتسمح باستخدام عرض النطاق الترددية من عدة رفوف عند قراءة البيانات. تقوم هذه السياسة بتوزيع النسخ المتماثلة بشكل متساو في العنقود مما يجعل من السهل تحقيق توازن الحمل. ولكنها تزيد من تكاليف الكتابة لأن الكتابة تحتاج لنقل الكتل إلى رفوف متعددة. لذلك في حالة الشائعة عندما يكون عامل النسخ المتماثل هو ثلاثة:

- توضع نسخة على عقدة في الرف المحلي.
- توضع نسخة أخرى على عقدة في رف بعيد مختلف.
- وتوضع الأخيرة في عقدة مختلفة في نفس الرف البعيد.

هذه السياسة لا تؤثر على موثوقية البيانات لأن فرصه فشل عقدة أكبر بكثير من فشل رف. يحاول HDFS تلبية طلب القراءة من النسخة الأقرب إلى القارئ وذلك لتقليل استهلاك عرض النطاق الترددية العام وتأخير القراءة. إذا كان هناك نسخة على نفس رف العقدة القارئة يفضل أن تكون هي النسخة التي تلبى طلب القراءة.

## 2-3 التسامح مع الفشل في هادوب ماب ريديوس :Hadoop MapReduce Fault Tolerance

في الحقيقة قد يحوي كود المستخدم أخطاء، وقد تتعطل العمليات، وتفشل الأجهزة، تتميز هادوب بقدرتها على التعامل مع مثل هذا الفشل والسماح للعمل أن يكتمل بنجاح رغم وجود الفشل، يشمل الفشل أي من الكيانات التالية :المهمة، مدير التطبيق، مدير العقدة، ومدير الموارد[3].

**3-2-1 فشل المهمة Task Failure :**الحالة الأولى لفشل المهمة والأكثر شيوعاً هو عندما يرمي كود المستخدم مهام المقابلة والاختزال (runtime exception) استثناء وقت التشغيل في حالة حدوث ذلك تقوم JVM الخاص فيها بارجاع الخطأ إلى مدير التطبيق الخاص بها قبل أن يخرج، ويتم تدوين الخطأ في سجلات(logs) المستخدم ثم يحدد مدير التطبيق المهمة كمهمة فاشلة (failed)، وتحرر الحاوية بحيث تكون مواردها متاحة لمهمة أخرى.

الحالة الثانية للفشل هو الخروج المفاجئ الخاص JVM ربما يكون هناك خطأ JVM يؤدي إلى إنهاء JVM من أجل مجموعة معينة من الظروف التي يتعرض لها كود مستخدم ماب ريديوس في هذه الحالة يلاحظ مدير العقدة أن العملية قد خرجت وبلغ مدير التطبيق الرئيسي ليحدد هذه المحاولة بأنها فاشلت.

يتم التعامل مع المهام المتعلقة بشكل مختلف حيث يلاحظ مدير التطبيق أنه لم يتلقى تحديداً للتقدم لبعض الوقت ويتتابع لتحديد المهمة على أنها فاشلة، وسيتم قتل عملية JVM الخاصة بالمهام تلقائياً بعد هذه الفترة، فترة المهلة التي تعتبر بعدها المهام الفاشلة هي عادة 10 دقائق ويمكن تهيئتها على أساس كل عمل (أو أساس مجموعة) عن طريق تعريف خاصية mapreduce.task.timeout إلى قيمة بالمili ثانية.

يؤدي إعطاء المهلة قيمة صفر إلى تعطيل المهلة بحيث لا يتم وضع علامة على المهام التي يتم تشغيلها منذ فترة طويلة على أنها فاشلة .في هذه الحالة لن يؤدي تعليق المهمة مطلقاً إلى تحرير الحاوية الخاصة بها، ومع مرور الوقت قد يحدث تباطؤ في العقود نتيجة لذلك لذلك ينبغي تجنب هذا النهج، والتتأكد من أن المهمة ترسل تقارير عن تقدمها بشكل دوري. عند إعلام مدير التطبيق أنَّ محاولة مهمة قد فشلت سيقوم بإعادة جدولة تنفيذ المهمة، سيحاول مدير التطبيق تجنب إعادة جدولة المهمة على مدير التطبيق مرة أخرى ويسأل عن مسبقاً، علاوة على ذلك إذا فشلت إحدى المهام أربع مرات، فلن يتم إعادة محاولة جدولتها مرة أخرى، هذه القيمة قابلة للإعداد حيث يتم التحكم في الحد الأقصى لعدد محاولات تشغيل مهمة بواسطة الخاصية mapreduce.map.maxattempts لمهمة المقابلة و mapreduce.reduce.maxattempts لمهمة الاختزال و بشكل افتراضي في حالة فشل أي مهمة أربع مرات سوف يفشل العمل بأكمله.

## 3-2-2 فشل مدير التطبيق Application Master Failure :

كما في مهام ماب ريديوس التي يتم إعطاؤها عدة محاولات لتحقيق النجاح (في مواجهة فشل الأجهزة أو الشبكة) يتم إعادة محاولة تشغيل التطبيقات في YARN في حالة حدوث فشل يتم التحكم في الحد الأقصى لعدد محاولات تشغيل مدير تطبيق ماب ريديوس بواسطة الخاصية mapreduce.am.max-attempt القيمة الافتراضية هي 2. لذلك إذا فشل مدير تطبيق رئيسي مرتين فلن يتم تجربته مرة أخرى وسيفشل العمل، تتم عملية الاسترداد كالتالي: يرسل مدير التطبيق نبضات دورية إلى مدير الموارد وفي حالة حدوث فشل في مدير التطبيق، سيقوم مدير الموارد باكتشاف الفشل وبعد نسخة جديدة من مدير التطبيق ضمن حاوية جديدة (تم إدارتها بواسطة مدير العقدة).

## 3-3-2 فشل مدير العقدة Node Manager Failure :

إذا فشل مدير العقدة عن طريق تعطله أو تشغيله ببطء شديد فسوف يتوقف عن إرسال ضربات القلب إلى مدير الموارد (أو إرسالها بشكل غير متكرر)، سيلاحظ مدير الموارد توقف مدير العقدة عن إرسال ضربات القلب إذا لم يستلم رسالة ضربة قلب لمدة عشر دقائق سيقوم بحذفه من قائمة العقد التي سيتم جدولته الحاويات عليها، سيتم استرداد أي مهمة أو مدير تطبيق يتم تشغيله على مدير العقدة الفاشلة باستخدام الآليات السابقة الذكر .بالإضافة إلى ذلك يرتب مدير التطبيق عملية إعادة تشغيل مهام المقابلة التي تم تشغيلها وإكمالها بنجاح على مدير العقدة الفاشلة ليتم إعادة تشغيلها إذا كانت تتتمى إلى أعمال غير مكتملة نظراً لأن مخرجاتها المرحلية الموجودة على نظام الملفات المحلي لمدير العقدة الفاشلة قد لا يمكن الوصول إليها من قبل مهام الاختزال، قد يتم إدراج مدير العقدة في القائمة السوداء(blacklisted) إذا كان عدد مرات فشل التطبيق مرتفعاً، حتى إذا لم يفشل مدير العقدة نفسه يتم إنجاز القائمة السوداء بواسطة مدير التطبيق.

#### **:Resource Manager Failure 4-3-2 فشل مدير الموارد**

يعتبر فشل مدير الموارد أمر خطير لأنه بدونه لا يمكن إطلاق الأعمال أو حاويات المهام قبل Hadoop 2.4 كان مدير الموارد يعتبر نقطة واحدة للفشل، نظراً لأنه في الحدث (غير المحتمل) لفشل الجهاز تفشل جميع الأعمال قيد التشغيل ولا يمكن استردادها. لتحقيق توفر عالي(HA) من الضروري تشغيل زوج من مدير الموارد أحدهما مستعد(standby) والآخر نشط(Active) إذا فشل مدير الموارد النشط، سيقوم المدير الاحتياطي بأخذ دوره بأسرع ما يمكن بحيث لا يحصل مقاطعة كبيرة للعميل.

يتم تخزين المعلومات حول جميع التطبيقات قيد التشغيل في مخزن حالة متاح بشكل كبير (مدعوم من قبل ZooKeeper أو HDFS) بحيث يمكن لمدير الموارد المستعد استرداد الحالة الأساسية لمدير الموارد النشطة الفاشلة لا يتم تخزين معلومات مدير العقد في مخزن الحالة(state store) نظراً لأنه يمكن إعادة بنائها بشكل سريع نسبياً بواسطة مدير الموارد الجديد عندما يرسل مدير العقد أول نبضاتهم. لاحظ أيضاً أن المهام ليست جزءاً من حالة مدير الموارد ، نظراً لأنه يتم إدارتها من قبل مدير التطبيق.



### الفصل الثالث

أباتشي سبارك

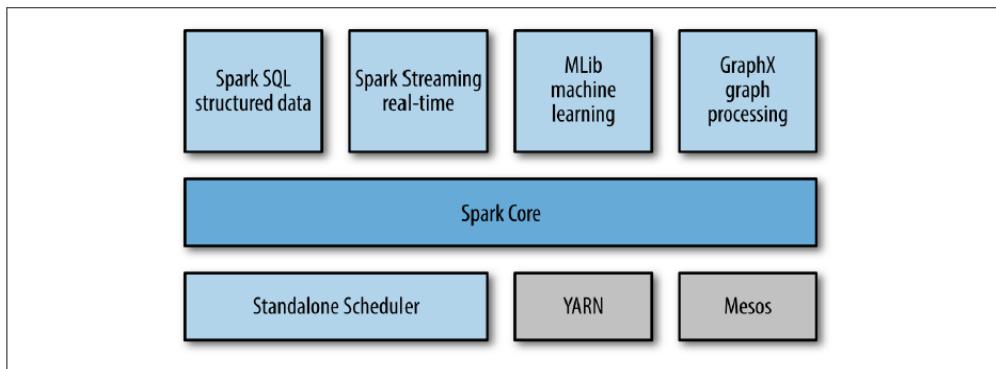
## Apache Spark

### 1-3 مقدمة [63]:Introduction

في وقت قصير جداً بُرِزَ أباتشي سبارك كجيل جديد لمعالجة البيانات الضخمة ويجري تطبيقه بشكل أسرع من أي وقت مضى . يُشَغِّل سبارك التطبيقات الدفعية batch applications التي يدعمها هادوب بالإضافة إلى دعمه مجموعة متنوعة من أعباء العمل بما في ذلك الاستعلامات التفاعلية (interactive queries) والبث التدفقية (streaming) والتعلم الآلي ومعالجة الرسوم البيانية. مع الارتفاع السريع في شعبية سبارك، يعد نقص المواد المرجعية الجيدة التي تتكلم عنه مصدر قلق كبير . يقدم سبارك ثلاثة فوائد رئيسية، أولًا إنه سهل الاستخدام ثانياً يعتبر سبارك سريع، ويمكن المستخدم من الاستخدام التفاعلي وتنفيذ خوارزميات معقدة، ثالثاً يعتبر سبارك منصة عمل عامة مما يتتيح الجمع بين أنواع متعددة من الحسابات (مثل استعلامات SQL ومعالجة النصوص والتعلم الآلي) التي كانت تتطلب في السابق منصات عمل مختلفة.

### 2-3 مكدس سبارك الموحد [59] [63] A Unified Stack of Spark

يحتوي مشروع سبارك على العديد من المكونات المتكاملة بشكل وثيق إذ يعتبر سبارك في نواته "محرك حاسوبي" مسؤول عن جدولة وتوزيع ومراقبة التطبيقات التي تتكون من العديد من المهام الحسابية عبر العديد من الأجهزة العاملة، يتسم سبارك في نواته بالسرعة والشموليّة من خلال احتواه على العديد من مكونات تشغّل أعباء عمل عالية المستوى مثل SQL أو التعلم الآلي. تم تصميم هذه المكونات للتّفاعل مع بعضها مما يتّيح للمستخدم الجمع بينها بشكل مشابه لعملية إضافة مكتبات إلى مشروع برمجيات، وفيما يلي مكونات سبارك بإيجاز:



الشكل(1-3) مكدس سبارك الموحد.[63].

1- **نواة سبارك Spark Core:** يحتوي على الوظائف الرئيسية لسبارك حيث تحتوي النواة مكونات من أجل جدولة المهام وإدارة الذاكرة والاسترداد من الأعطال والتفاعل مع أنظمة التخزين، كما تحتوي نواة سبارك على مجموعة من الـ API والتي تحتوي تعريف الوحدة البنائية لسبارك (Resilient Distributed Data- sets RDDs) وهي مجموعة من العناصر الموزعة على عدة عقد والتي يتم معالجتها على التوازي.

2- **Spark SQL:** هو حزمة مخصصة للتعامل مع البيانات المهيكلة structured data ويمكن المطورين من عملية دمج استعلامات القواعد البيانات sql والعمليات على ال RDD.

3- **Spark Streaming:** هو مكون سبارك الذي يتيح معالجة البث المباشر للبيانات مثل ملفات السجل log files يتم إنشاؤها بواسطة خوادم ويب أو رتل الرسائل التي تحتوي على تحديثات الحالة التي تم نشرها بواسطة مستخدمي خدمة ويب. يوفر Spark Streaming واجهة برمجة تطبيقات لمعالجة تدفقات البيانات التي تتطابق إلى حد كبير مع واجهة برمجة التطبيقات الخاصة بـ Spark Core RDD، مما يسهل على المبرمجين معرفة المشروع والانتقال بين

التطبيقات التي تعالج البيانات المخزنة في الذاكرة أو على القرص أو البيانات التي تصل في الوقت الفعلي. تم تصميم **Spark Streaming** ل توفير نفس الدرجة من التسامح مع الفشل (fault tolerance) والقدرة الإنتاجية وقابلية التوسيع المتوافرة في نواة سبارك.

- 4 **MLib**: مكتبة سبارك من أجل دعم خوارزميات التعلم الآلي (ML machine learning algorithms) وأنواعاً متعددة من خوارزميات التعلم الآلي ، بما في ذلك التصنيف والانحدار والتجميع والترشيح التعاوني ، بالإضافة إلى وظائف الدعم مثل تقييم النماذج واستيراد البيانات.
- 5 **GraphX**: مكتبات متخصصة بالمعالجة الموزعة على المخططات (graphs).

-6 **مدير العقود Cluster Managers** [59] : صُمم سبارك لتعمل بشكل فعال على عنقود مؤلف من واحد إلى عدةآلاف من العقد الحاسوبية، ولتحقيق هذا الأمر مع زيادة المرونة إلى أقصى حد ممكن، يمكن لشركة سبارك تشغيل مجموعة متعددة من مدير العقائد بما في ذلك YARN ويعتبر خياراً مناسباً إذا كان هادوب مثبت مسبقاً وApache Mesos إذا كان نملك أباتشي ميسوس مسبقاً على العقد و مدير عنقود بسيط مدرج في سبارك نفسه يُطلق عليه Standalone Scheduler ويعتبر مناسباً إذا تم تثبيت سبارك على مجموعة من الأجهزة الفارغة[63][35].

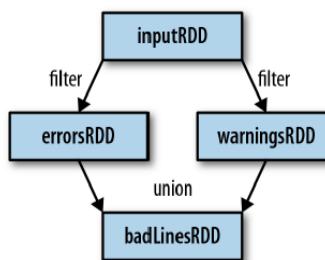
### 3-3 طبقات تخزين البيانات في سبارك :Storage Layers for Spark

يمكن لسبارك إنشاءمجموعات بيانات موزعة من أي ملف مخزن في نظام الملفات الموزع (HDFS) أو أنظمة تخزين أخرى (Amazon S3، HBase، Hive، Cassandra، Apache Mesos ) ، وما إلى ذلك). من المهم أن نؤكد أن سبارك لا يتطلب هادوب لأنه وببساطة لديه دعم لأنظمة تخزين أخرى غير مدعومة من قبل هادوب فهو يدعم ملفات نصية، ملفات تسلسلية، Parquet، Avro .. الخ..

### 4-3 Resilient Distributed Dataset (RDD)

إن RDD هي بنية البيانات الأساسية لApache Spark وهي ببساطة مجموعة موزعة من العناصر الثابتة وغير قابلة للتعديل التي تقسم منطقياً وتحسب على العقد المختلفة في العقد، يتم التعبير عن جميع الأعمال إما بإنشاء RDD جديدة أو تحويل RDDs موجودة إلى أخرى أو استدعاء عمليات على RDDs لحساب نتيجة، ينشئ المستخدمون RDDs عن طريق تحميل مجموعة بيانات خارجية في برنامج التشغيل (driver program) الخاص بهم وتقوم سبارك تلقائياً ب التقسيم كل RDD إلى أقسام متعددة وتوزيع البيانات الموجودة في RDDs عبر العقد وتوالي العمليات التي تقوم بها عليها. بمجرد إنشائها يمكن القيام عليها ب نوعين من العمليات: التحويلات (transformations) والإجراءات (actions). تقوم التحويلات بإنشاء RDD جديدة من أخرى سابقة من خلال اجراء تعديلات عليها (إحدى عمليات التحويل الشائعة هي تصفية البيانات التي تطابق القيمة التي يتم تمريرها إلى دالة التقييم) مثل يمكننا استخدام التحويل filter() لإنشاء RDD جديدة تحمل فقط السلاسل التي تحتوي على كلمة معينة.

يتم إعادة حساب RDDs افتراضياً في كل مرة يتم تشغيل إجراء (action) عليها، لذلك عند الحاجة إلى إعادة استخدام RDD في إجراءات متعددة، يمكن الاحتفاظ بها في أماكن مختلفة يحددها المستخدم وذلك عن طريق الدالة persist(). . بعد حسابها للمرة الأولى، ستخزن سبارك محتويات RDD في الذاكرة مقسمة عبر الأجهزة في العقد. كما يمكن أيضاً حفظ الـ RDDs على القرص بدلاً من الذاكرة قد يبدو سلوك عدم الاحتفاظ بالـ RDDs بشكل افتراضي مفيد عند التعامل معمجموعات البيانات الضخمة. عندما تستتبّط RDDs جديدة من بعضها البعض باستخدام التحويلات، فإن سبارك يقوم بتتبع مجموعة التبعيات بين الـ RDDs المختلفة، والتي تدعى الرسم البياني للنسب (Lineage graph) ويستخدم هذه المعلومات لحساب كل RDD حسب الطلب واستعادة البيانات المفقودة إذا ما فقد جزء من RDD يوضح الشكل (2-5) رسم بياني للنسب:



الشكل (3-2) مثال على رسم بياني للنسب [63].

تحقق الـ RDD تسامحاً جزئياً مع الأخطاء وذلك من خلال خاصية تتبع نسب التحويلات المطبقة على الـ RDD حيث يتم تسجيل التحويلات المطبقة على RDD ليتم تكرار تنفيذها عند فقد البيانات أو فشل عقد العنقود، وتحقق كفاءة عالية من خلال موازاة المعالجة عبر العقد المتعددة في العنقود، وتقليل تكرار البيانات بين تلك العقد.

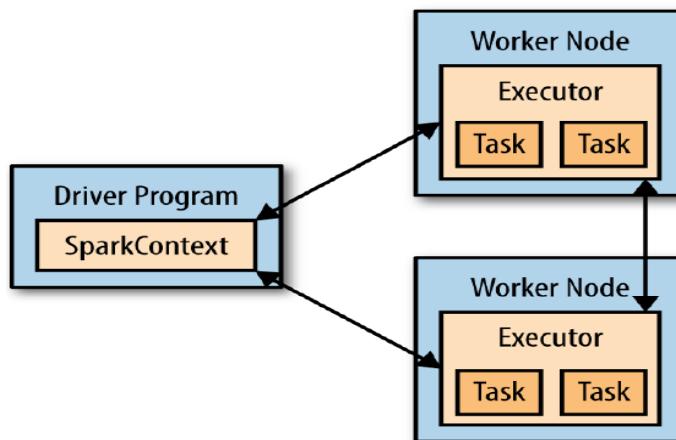
### 3-4-3 التقييم الكسول [Lazy Evaluation]:

يتم تقييم التحويلات على الـ RDDs بطريقة كسولة وهذا يعني أنه لن يتم تنفيذها حتى يحتاج الإجراء اللاحق إلى النتيجة الخاصة به. يعني التقييم الكسول أنه عندما نستدعي تحويل على RDD (على سبيل المثال، استدعاء التحويل () لا يتم تنفيذ العملية على الفور، بدلاً من ذلك يقوم سبارك داخلياً بتسجيل البيانات الوصفية التي تدل على أنه قد تم طلب هذا التحويل على هذه الـ RDD ، أي بدلاً من الاحتفاظ بالبيانات التي تحتويها RDD فإنه يتم الاحتفاظ بكيفية أو ما هي التحويلات اللازمة لحساب البيانات مما يؤدي إلى تجنب الحاجة إلى معالجة البيانات دون داعٍ كذلك الأمر بالنسبة لعملية تحميل البيانات إلى RDD فعند استدعاء () لا يتم تحميل البيانات حتى تكون ضرورية، تستخدم "Spark" تقييماً كسولاً لتقليل عدد مرات الوصول إلى البيانات من خلال تجميع العمليات معاً، في الأنظمة مثل هادوب غالباً ما يسيطر المطوروون إلى قضاء الكثير من الوقت في التفكير في كيفية تجميع العمليات معاً لتقليل عدد مرات وصول ماب ريديوس إلى البيانات، أما في سبارك بدلاً من كتابة مهمة مقابلة معقدة واحدة يمكن ربط العديد من العمليات البسيطة، وبالتالي يستطيع المستخدمون بحرية تنظيم برامجهم إلى عمليات أصغر وأكثر قابلية للإدارة.

### 3-5 التنفيذ الموزع في سبارك [Distributed execution in Spark]:

يتكون العمل في سبارك من مخطط غير دوري مباشر (DAG) لمجموعة من المراحل يكافئ كل منها تقريباً طور مقابلة أو اختزال في ماب ريديوس، يتم تقسيم المراحل إلى مهام ويتم تشغيلها بالتوالي على أجزاء من RDD موزعة عبر العنقود — تماماً مثل المهام في ماب ريديوس - يتم تشغيل العمل على هيئة تطبيق (يمثل بنسخة من SparkContext) ويمكن للتطبيق تشغيل أكثر من عمل واحد، بشكل تسلسلي أو بالتوالي، ويوفر آلية لوصول العمل إلى RDD التي تم تخزينها مؤقتاً بواسطة عمل سابق في نفس التطبيق.

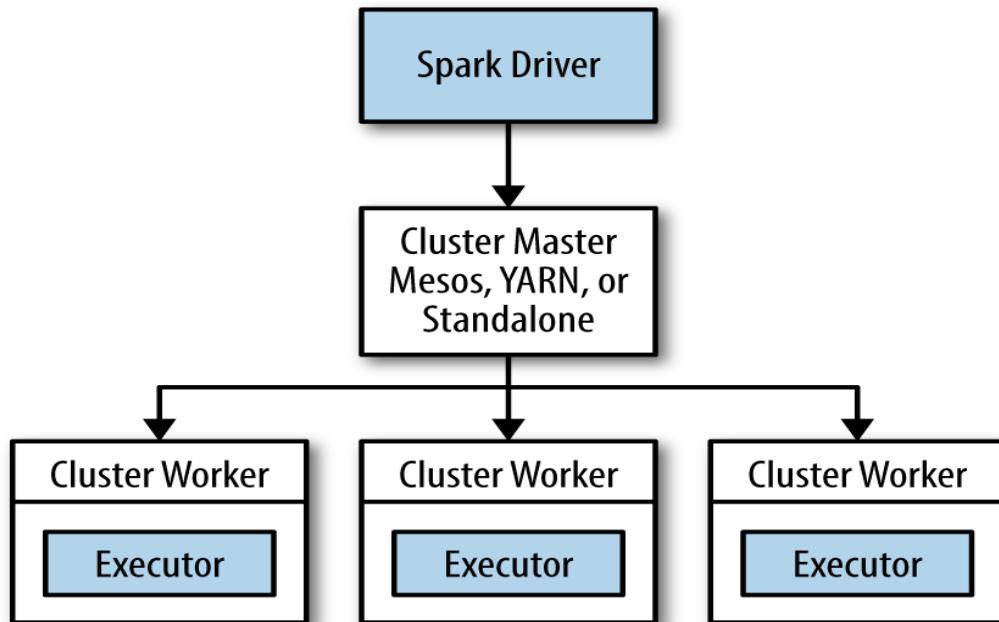
يتكون كل تطبيق سبارك من برنامج تشغيل driver program يطلق عمليات متوازية مختلفة على العنقود، يحتوي برنامج التشغيل على التابع الرئيسي للتطبيق (application's main function) ويحدد مجموعات البيانات الموزعة على العنقود، ثم يطبق العمليات عليها، يصل برنامج التشغيل إلى سبارك من خلال الغرض المدعو SparkContext الذي يمثل الاتصال مع العنقود ونستطيع من خلاله بناء RDD جديدة، ولتشغيل عمليات على RDD يقوم برنامج التشغيل عادة بإدارة عدد من العقد تسمى المنفذين (executors) ، على سبيل المثال إذا كنا نقوم بتشغيل عملية () على العنقود ، فقد تقوم أجهزة مختلفة بإجراء الحساب على سطور في نطاقات مختلفة من الملف، تأخذ سبارك التابع المراد تنفيذه تلقائياً وترسله إلى العقد المنفذة، وبالتالي يمكنك كتابة التعليمات البرمجية في برنامج تشغيل واحد ويتم تلقائياً تشغيل أجزاء منه على عقد متعددة، يوضح الشكل (3-5) كيفية عمل سبارك.



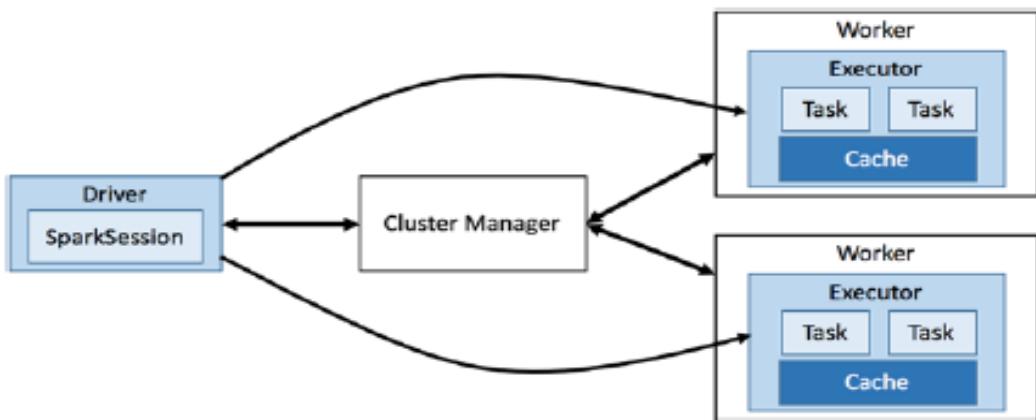
الشكل(3-3) مكونات المعالجة الموزعة في سبارك[63].

### 6-3 مكونات عنقود سبارك : [63] [59] Spark Cluster Components

يستخدم سبارك في نمط التنفيذ الموزع بنية (السيد / العبد) مع منسق مرکزي واحد والعديد من عقد العمال الموزعة يسمى المنسق المرکزي بالمشغل (Driver). يتواصل المشغل مع عدد كبير من العمال الموزعين يطلق على كل منهم اسم المُنْقَد، يتم تشغيل برنامج المشغل في عملية جافا خاصة به وكل مُنْقَد يعمل في عملية جافا منفصلة أيضاً، يشكل كل من المشغل والمنفذين الخاصين به تطبيق سبارك السالف الذكر، يتم إطلاق تطبيق سبارك على مجموعة من الأجهزة باستخدام خدمة خارجية تسمى مدير العنقود الذي يعرف العمال ومكان توضعهم وحجم الذاكرة وعدد الأنوية لدى كل عقدة عاملة.



الشكل(4-3) مكونات تطبيق سبارك الموزع[63].



الشكل(5-3) آلية تنفيذ سبارك للتطبيق الموزع[59].

### 1-6-3 المشغل [The Driver]: [59][63]

المُشغّل هو العملية التي يتم فيها تشغيل التابع main إنها العملية التي تقوم بتشغيل كود المستخدم الذي يقوم بتكوين RDDs ويقوم بتنفيذ التحويلات والإجراءات وبمجرد انتهاء المشغل يتم إنتهاء التطبيق. عندما يُشغّل المشغل يقوم بواجبين:

1. **تحويل برنامج المستخدم إلى مهام**[59]: يعتبر برنامج سبارك مسؤولاً عن تحويل برنامج المستخدم إلى وحدات للتنفيذ الفعلي تسمى المهام. وتتبع جميع برامج سبارك البنية نفسها: فهي تتكون من بعض المدخلات، ويشتق RDDs جديدة من تلك المنشأة باستخدام التحويلات وتتفق إجراءات لجمع أو حفظ البيانات، ينشئ برنامج سبارك رسمًا بيانيًا منطقيًا directed acyclic graph (DAG) من العمليات، عند تشغيل برنامج التشغيل يقوم بتحويل هذا الرسم البياني المنطقي إلى خطة تنفيذ فعلية ويحول سبارك (DAG) إلى مجموعة من المراحل، كل مرحلة بدورها تتكون من مهام متعددة ويتم تجميع المهام وتحضيرها لإرسالها إلى العنقود، تعتبر المهام أصغر وحدة عمل في سبارك، ويمكن لبرنامج المستخدم النموذجي إطلاق المئات أو الآلاف من المهام الفردية.

2. **جدولة المهام على المنفذين**: بالاعتماد على خطة التنفيذ الفعلي يجب على المشغل تنسيق جدولة المهام الفردية على المنفذين حيث عند بدء تشغيل المنفذين يطلب المشغل حجز موارد للمنفذين على العقد العاملة واطلاقهم ثم يقوم المنفذون بتسجيل أنفسهم مع المشغل بحيث يكون لديه رؤية كاملة عن منفذ التطبيق في جميع الأوقات. يمثل كل مُنْفَذ (executor) عملية قادرة على تشغيل المهام وتخزين بيانات الـ RDD سيقوم برنامج تشغيل سبارك بالنظر في المجموعة الحالية من المنفذين ومحاولة جدولة كل مهمة في موقع مناسب بناءً على موضع البيانات. قد يتتأثر تنفيذ المهام بالبيانات المخزنة مؤقتاً لذلك يتبع برنامج التشغيل أيضًا موقع البيانات المخزنة مؤقتاً ويستخدمها لجدولة المهام المستقبلية التي تصل إلى تلك البيانات.

### 2-6-3 المنفذين [Executors]: [59]

إن المنفذين في سبارك هي عمليات (processes) عاملة مسؤولة عن تشغيل المهام الفردية في عمل سبارك يتم تشغيل المنفذين مرة واحدة في بداية تطبيق سبارك وعادة ما يتم تشغيله طوال عمر التطبيق، على الرغم من أن تطبيقات سبارك يمكن أن تستمر إذا فشل المنفذون.

للمنفذين دوران، أو لا يقوم المنفذون بتشغيل المهام التي تشكل التطبيق وإرجاع النتائج إلى المشغل. ثانياً، توفر التخزين في الذاكرة لـ RDDs التي يتم تخزينها مؤقتاً بواسطة برنامج المستخدم، من خلال خدمة تسمى Block Manager التي تعيش داخل كل مُنْفَذ، ونظرًا لأن RDDs يتم تخزينها مؤقتاً بشكل مباشر داخل المنفذين يمكن تشغيل المهام جنبًا إلى جنب مع البيانات المخزنة مؤقتاً.

## [3]:Spark on YARN 7-3

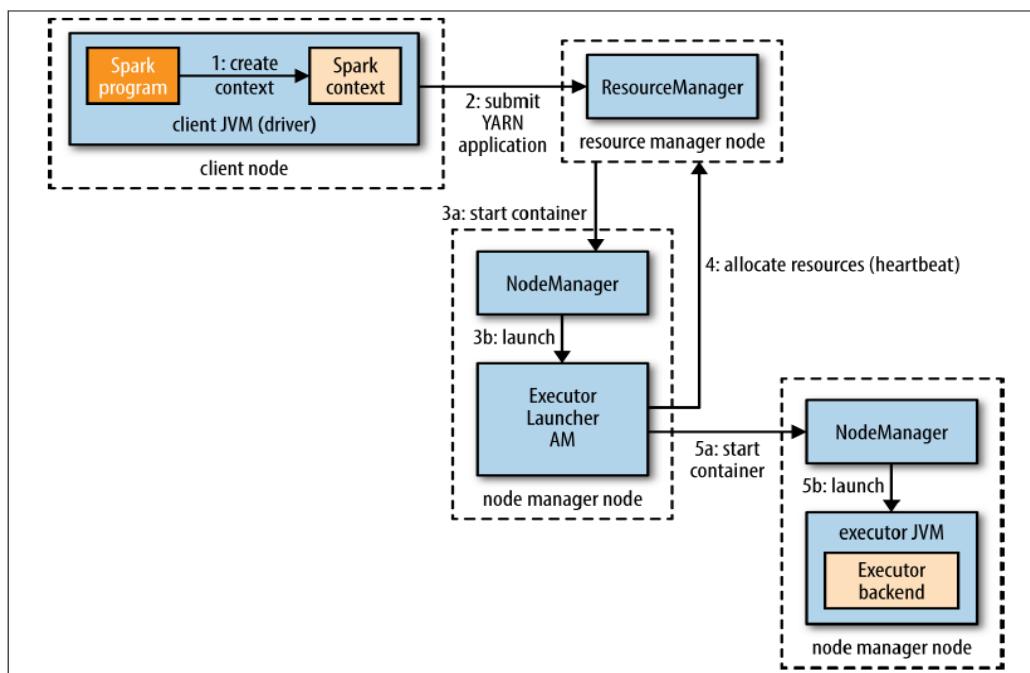
إن تشغيل سبارك على YARN يوفر تكامل مع مكونات هادوب الأخرى وهو الطريقة الأكثر ملائمة لاستخدام سبارك عندما يكون لديك نظام هادوب موجود بالفعل، توفر سبارك وضعی نشر ليتم تشغيلها على YARN:

### ► [3] YARN client mode

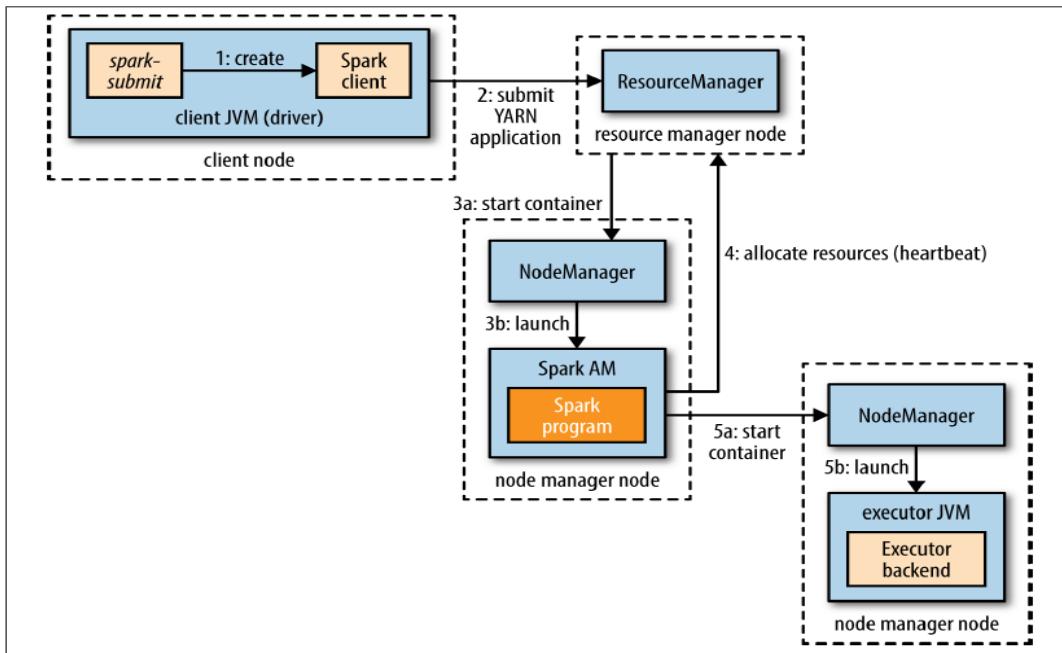
يعتبر هذا النمط مناسب للبرامج التي تحتاج إلى تفاعل بيداً yarn عندما يتم بناء SparkContext جديد من قبل مشغل البرنامج driver program الذي يقوم بإرسال تطبيق yarn إلى مدير الموارد الذي بيداً حاوية جديدة على مدير عقدة في العقدود ويشغل عليها ExecutorLauncher AM الذي يتمثل عمله بإطلاق processesExecutorBackend ضمن حاويات yarn من خلال طلبه موارد جديدة من قبل مدير العقدود وإطلاق processesExecutorBackend ضمن الحاويات المحوسبة، عندما بيداً الـ executor يقوم بتسجيل نفسه مع SparkContext مما يساعد على معرفة عدد المنفذين المتاحين لتشغيل المهام عليهم وأماكن تواجدهم، إن عدد المنفذين الافتراضي هو 2 وعدد الأنوية المتاحة لكل منهم هو واحد ومقدار الذاكرة الافتراضي هو 1024 ميغابايت ويمكن تغيير القيم من خلال بارامترات التعليمية -spark submit أثناء تشغيل البرنامج.

### ► [3] YARN cluster mode

في هذا النمط يتم تشغيل برنامج سبارك driver program داخل العملية المخصصة لمدير تطبيق yarn وليس على عقدة الزبون كما في النمط السابق، فالفرق الجوهرى بين النمطين يمكن بمكان تشغيل Spark driver ففي نمط الزبون يتم تشغيله على عقدة الزبون أما في نمط العقدود يتم كبسلة الـ Spark driver داخل YARN Application Master.



الشكل(6-3) نمط النشر [3] Client Mode



الشكل(3-7) نمط النشر .[3] Cluster Mode



## الفصل الرابع

### التسامح مع الأعطال في سبارك

#### Apache Spark Fault Tolerance

##### 1-4 مقدمة [51]:

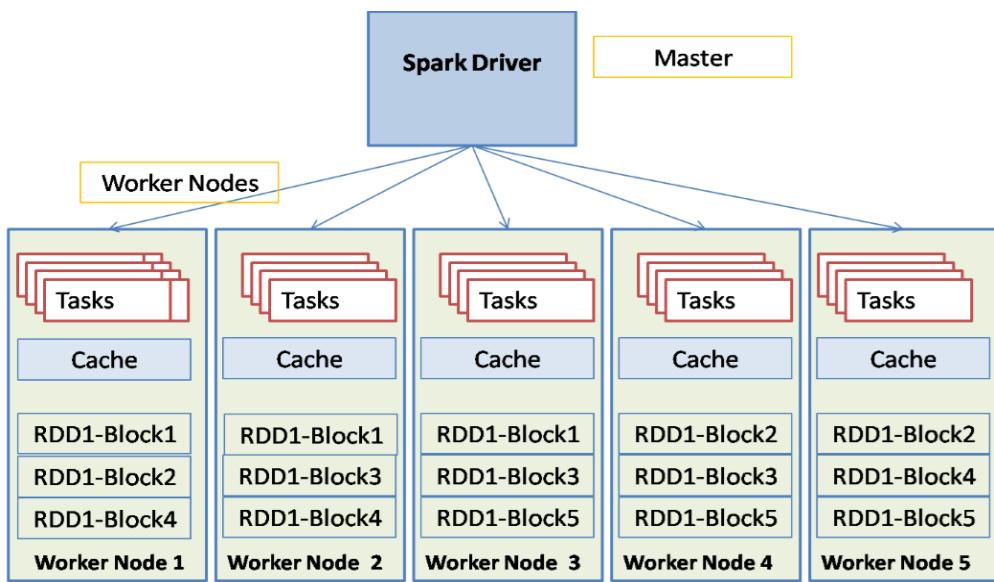
تم اعتماد نماذج البرمجة العنقودية عالية المستوى مثل ماب ريديوس على نطاق واسع لمعالجة الكميّات المتزايدة من البيانات في الصناعة والعلوم، تعمل هذه الأنظمة على تبسيط البرمجة الموزعة عن طريق توفير جدولة مدركة لمكان توضع البيانات locality-aware scheduling و التسامح مع الفشل و موازنة الحمل مما يتيح لمجموعة واسعة من المستخدمين تحليلاً مجموعات البيانات الضخمة الخاصة بهم باستخدام عناقيد من الأجهزة السلعية، تعتمد معظم أنظمة الحوسبة العنقودية الحالية على نموذج تدفق بيانات لا حلقي (acyclic data flow model) أي يتم تحميل السجلات من التخزين الدائم (على سبيل المثال نظام ملفات موزع) يتم تمريرها من خلال DAG مكون من عمليات محددة ثم تعود إلى التخزين الدائم، تتيح معرفة الرسم البياني لتدفق البيانات جدولة تلقائية للعمل والتغافي من حالات الفشل.

##### 2-4 التسامح مع الأعطال في سبارك [74]:

إن إحدى المزايا الرئيسية في سبارك هي أنه يوفر ضمانات قوية للفشل إذ طالما يتم تخزين البيانات المدخلة بشكل موثوق فإن سبارك ستحسب دائماً النتيجة الصحيحة منها رغم حصول الفشل.

##### 1-2-4 التسامح مع الفشل الذي تقدمه [51] Resilient Distributed Datasets (RDDs):

تسمح مجموعات البيانات الموزعة المرنة (RDDs) للمبرمجين بإجراء عمليات حسابية في الذاكرة في العناقيد الحاسوبية الكبيرة الحجم مع الحفاظ على التسامح مع الفشل، تعد خاصية التسامح مع الفشل من بين الخصائص المرغوبة جداً، ولكن أكثرها صعوبة في التحقيق، بشكل عام هناك خياران لجعل مجموعة البيانات الموزعة متسامحة مع الفشل وهي إنشاء نقاط استعادة من البيانات أو تسجيل التحداثيات التي تم إجراؤها عليها، في بيئتنا المستهدفة فإن إنشاء نقاط استعادة للبيانات أمر مكلف: فهو يتطلب تكرار مجموعات البيانات الكبيرة عبر شبكة مراكز البيانات، والتي عادة ما يكون عرض نطاقها أقل بكثير من عرض نطاق الذاكرة داخل جهاز سيسكلهك أيضاً مساحات تخزين إضافية (يؤدي نسخ البيانات في ذاكرة الوصول العشوائي RAM إلى تقليل الكمية الكلية التي يمكن تخزينها مؤقتاً بينما يؤدي الدخول إلى القرص إلى إبطاء التطبيقات)، والتحويلات الخشنة coarse-grained transformations فقط بمعنى أنه يمكننا تسجيل عملية واحدة ليتم تطبيقها على العديد من السجلات، ثم نذكر سلسلة التحويلات المستخدمة لبناء RDD (أي النسب الخاص بها) واستخدامها لاستعادة الأجزاء المفقودة، وهذا ما تتبعه RDD لتحقيق التسامح مع الفشل بكفاءة حيث توفر شكلاً مقيداً للذاكرة المشتركة استناداً إلى تحويلات حتمية من نوع coarse-grained بدلاً من التحداثيات fine-grained للحالة المشتركة. كما تقوم RDD بإعادة بناء الأجزاء المفقودة من خلال النسب: يحتوي RDD على معلومات كافية حول كيفية استيفائه من RDDs أخرى لإعادة بناء القسم المفقود فقط، دون الحاجة إلى نقطة استعادة، تلعب البيانات المكررة دوراً مهمًا في عملية الاسترداد الذاتي إذ يمكننا استرداد البيانات المفقودة من خلال البيانات المكررة.



الشكل(4) تكرار RDD على العقد العاملة [75].

كيف يتم تحقيق التسامح مع الفشل من خلال DAG؟ كما نعلم أن DAG تحفظ بسجل العمليات المطبقة على RDD أي أنها تحفظ بكل التفاصيل عن المهام المنفذة على الأقسام المختلفة من ال RDD لذلك في حال الفشل أو في حالة فقدان أي RDD يمكننا جلبها بسهولة بمساعدة الرسم البياني DAG ، على سبيل المثال إذا كانت هناك أي عملية قيد التنفيذ وفجأة حدث فقد لبيانات RDD فإنه وبمساعدة مدير العنقود سوف نحدد القسم الذي حدث فيه الخسارة، بعد ذلك من خلال DAG سوف نSEND العمل إلى عقد جديدة التي ستعيد تنفيذ سلسلة العمليات اللازم للحصول على الجزء المفقود.

#### 2-2-4 نقاط التفتيش : [63][51] Check pointing :

على الرغم من أن معلومات النسب التي تتبعها ال RDDs تسمح دائمًا للبرنامج بالتعافي من حالات الفشل إلا أن هذا الاسترداد قد يكون مستهلكًا للوقت بالنسبة لـ RDDs ذات سلاسل النسب الطويلة، على سبيل المثال في المهام التي تعتمد على قيمة سابقة تكون من المفيد تخزين ال RDDs عند نقاط زمنية معينة على التخزين الدائم.

تعمل هذه الميزة بحفظ بيانات حول التطبيق بشكل دوري إلى نظام تخزين موثوق مثل Amazon S3 أو HDFS أو لاستخدامها في الاسترداد من الفشل، تقييد نقاط التفتيش في الحد من الحالات التي يجب إعادة حسابها عند الفشل عند فقد البيانات يتم إعادة حساب الحالة باستخدام الرسم البياني للنسب الخاص بالتحويلات، لكن نقاط التفتيش تتحكم في مدى الوقت الذي يجب أن نعود في الحساب إليه. يوفر سبارك حاليًا واجهة برمجة تطبيقات (API) للتحقيق خاصية نقطة الاستعادة check pointing حيث يتم تحقيق إجراء فحص لإنشاء نقاط استعادة بشكل تلقائي، نظرًا لأن المجدول يعرف حجم كل مجموعة بيانات بالإضافة إلى الوقت الذي استغرقه لمعالجتها، فإنه ينبغي أن يكون قادرًا على تحديد مجموعة مماثلة من RDDs لإنشاء نقاط استعادة عليها لتقليل وقت استرداد النظام.

#### 2-2-4 التسامح مع فشل المشغل : [63] Driver Fault Tolerance :

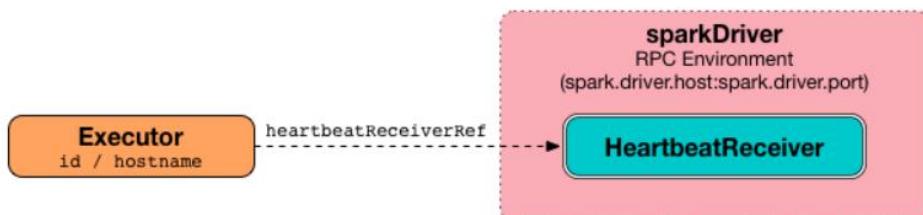
عندما يفشل المشغل في البرنامج الدفعية سيتم إنهاء المنفذين ويفشل تطبيق سبارك بالكامل، حيث لا تفقد RDDs نفسها فحسب بل يفقد العاملون أيضًا سيدهم الذي يتلقون الأوامر منه، وسيتم فقد جميع نتائج الحسابات الحالية والتي أكملاها العمال أيضًا، حيث يترك المستخدم فشل المشغل بدون أي حل سوى إعادة تشغيل التطبيق من البداية .

تطلب حالات الفشل في عقدة برنامج التشغيل طريقة خاصة لإنشاء SparkContext بدلًا من إنشاء SparkContext جديد يتم استدعاء () SparkContext.getOrCreate() الذي يأخذ الدليل الذي قامت نقاط الاستعادة بوضع البيانات فيه. لا تقوم سبارك تلقائيًا بإعادة تشغيل المشغل في حالة تعطلها، لذلك تحتاج إلى مراقبته باستخدام أداة مثل monit وإعادة تشغيله.

: [63][51] Worker Fault Tolerance 4-2-4 التسامح مع فشل العقدة العاملة

يتم إنشاء الرسم البياني للنسب (DAG) وإدارة تنفيذه من قبل مشغل البرنامج سيعالج العمال جزءاً من الـ DAG وسيقومون بإرسال نتائج المهام إلى مشغل البرنامج الذي يقوم بتتبع المهام ومعلومات الكتل الخاصة بالعقد العاملة مما يعني أنه في حالة فشل العمال، فإنه يمكن إعادة جدولة المهام على العمال الآخرين، علماً أنه سيتم فقدان نتائج الحسابات الغير مرسلة إلى المشغل، من أجل التسامح مع فشل العقدة العاملة يقوم سبارك بتكرار جميع البيانات على العقد العاملة، وبالتالي جميع RDDs التي تم إنشاؤها من خلال التحويلات على بيانات الدخل المكررة تتسامح مع فشل العقدة العاملة، لأن خط النسب الخاص بالـ RDD يسمح للنظام بإعادة حساب البيانات المفقودة على طول الطريق من النسخة المتماثلة الباقية من البيانات المدخلة.

**5-4 التسامح مع فشل المُنفذ Executer Fault Tolerance :** كل مُنفذ يرسل رسالة نبضة إلى المشغل بشكل دوري كل 10 ثواني بشكل افتراضي ، عند اكتشاف فشل المُنفذ يقوم المشغل بإخبار مجدول المهام عن فقد المُنفذ الذي يقوم في وقت لاحق بمعالجة فقدان المهام المُنفذة على المُنفذ كما يخبر المشغل الـ DAG scheduler ليزيل كافة الآثار (مثل الخاصة بالمنفذ المفقود) shuffle blocks) على ذلك يتطلب المشغل أيضاً من SparkContext أن يستبدل المُنفذ المفقود باخر من خلال اتصاله مع مدير العقدود، عند فشل المُنفذ في البرامج الدفعية سيتم فقدان البيانات المخزنة مؤقتاً والـ RDDs المحسوبة جزئياً مع ذلك فإن Spark RDDs متسامحة مع هذا الفشل وستبدأ سبارك منفذًّا جديداً لإعادة حساب هذه البيانات من مصدر البيانات الأصلي سينخفض الأداء عند إعادة حساب البيانات ولكن فشل المُنفذ لن يؤدي إلى فشل العمل.



الشكل(4-2) آلية ضربات القلب في سيارك[75].

#### 6-2-4 ضمانات المعالجة :[75] Processing Guarantees

نظرًا لضمادات التسامح مع فشل العمال من سبارك يمكنها توفير دلالات (exactly once semantics) لكل التحويلات بمعنى أنه حتى إذا ما فشل أحد العمال وتم إعادة معالجة بعض البيانات، فإن النتيجة النهائية المحولة (أي RDDs) ستكون هي نفسها كما لو كانت البيانات تمت معالجتها مرة واحدة بالضبط.

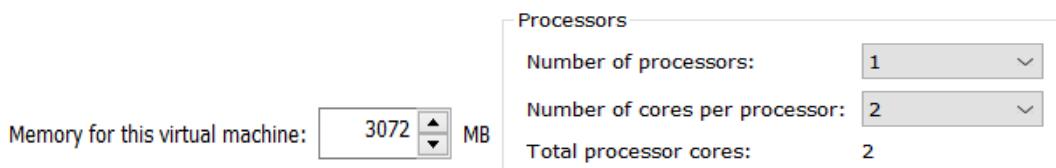


## الفصل الخامس

### القسم العملي

**1-5 مقدمة:** سنقوم بنشر هادوب وسبارك على أجهزة افتراضية للاستفادة من جميع مزاياها الافتراضية، والتي يمكن أن تجعل من إدارة العنقود أسهل، كما أن الأجهزة الافتراضية سريعة النشر ويمكن نقلها دون مخاطر [25] ، سنستخدم في بحثنا برنامج VMware وسيحتوي العنقود في كل من هادوب وسبارك على أربع عقد تمتلك كل منها الموصفات التالية:

Operating System: Ubuntu 16.04 - Memory: 3GB – Processors: 2 core -Hard Disk: 60 GB SSD.



1. العقدة master ستكون العقدة الرئيسية (192.168.65.157).
2. العقدة hdslave1 عقدة البيانات الأولى (192.168.65.156).
3. العقدة hdslave2 عقدة البيانات الثانية (192.168.65.155).
4. العقدة hdslave3 عقدة البيانات الثالثة (192.168.65.161).

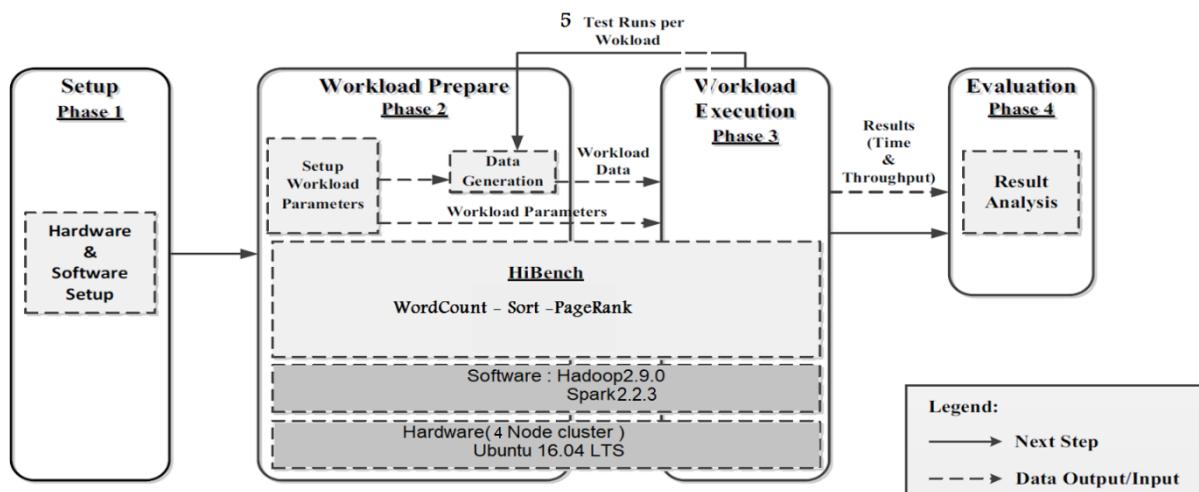
### 2 مواصفات جهاز الحاسوب المستخدم:

1. Processor :Intel® Core™ i7 7500U Processor, 2.7 GHz (4 M Cache, up to 3.5 GHz)
2. Operating System: Windows 10 Pro.
3. Memory: 16GB DDR4, 2133MHz.
4. Graphic: AMD + RadeonTM R5 M420 , with 2GB, VRAM
5. Storage: SATA HDD 1TB 5400RPM + SSD Hard Drive 240 GB Internal.

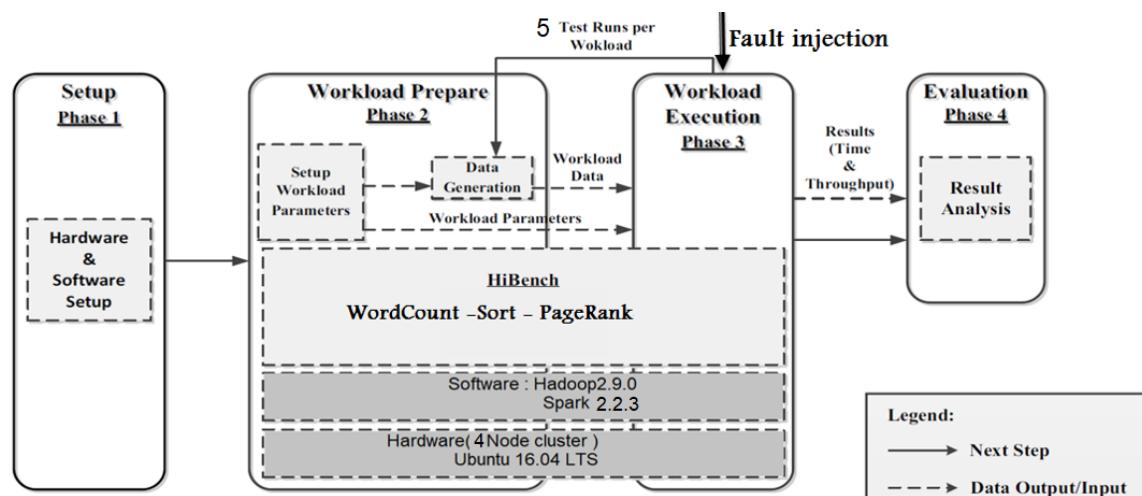
### 3 خطوات التنفيذ:

1. تثبيت هادوب.
2. تثبيت HiBench وربطه مع هادوب.
3. اختيار مجموعة من البرامج لاختبار الفشل أثناء تنفيذها.
4. تحديد المعايير المستخدمة لتقدير الأداء.
5. تنفيذ وتقدير أداء هادوب عند تنفيذ البرامج المختارة.
6. اختيار مجموعة من حالات الفشل الممكنة في كلا المنصتين.
7. تنفيذ وتقدير أداء هادوب عند تنفيذ البرامج مع حقن حالات الفشل في مكونات البيئة.
8. تثبيت سبارك وربطه مع HiBench.
9. تنفيذ وتقدير أداء سبارك عند البرامج المختارة.
10. تنفيذ وتقدير أداء سبارك عند تنفيذ البرامج مع حقن حالات الفشل في مكونات البيئة.
11. النتائج والتوصيات.

► وتقسم الخطوات السابقة إلى قسمين موضوعين بالشكلين التاليين:



الشكل(1-5) مخطط للقسم الأول لعملية المقارنة باستخدام HiBench



الشكل(2-5) مخطط للقسم الثاني لعملية المقارنة باستخدام HiBench.

### 3-1 تثبيت عنقود هادوب[1][64]

- تثبيت جافا على كل العقد من خلال التعليمية التالية:  
`sudo apt-get install openjdk-8-jdk`
- تثبيت بروتوكول ssh على كل العقد من خلال التعليمية التالية:  
`sudo apt-get install openssh-server`
- تحرير ملف hosts في جميع العقد من خلال التعليمية `sudo gedit /etc/hosts` وإدخال السطور التالية:

```
192.168.65.157 master
192.168.65.156 hdslave1
192.168.65.155 hdslave2
192.168.65.161 hdslave3
```

- تحرير ملف hostname في جميع العقد ووضع اسم العقد الجديد
  - توليد مفتاح ssh على العقد الرئيسية : `.ssh-keygen`
  - نسخ محتوى الملف `ssh/id_rsa.pub` في العقد الرئيسية إلى `ssh/authorized_keys` في العقد الرئيسية:
- ```
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```
- وكذلك إلى جميع العقد الأخرى من خلال التعليمات التالية:

```
$ sudo ssh-copy-id -i /home/manal/.ssh/id_rsa.pub manal@hdslave1
$ sudo ssh-copy-id -i /home/manal/.ssh/id_rsa.pub manal@hdslave2
```

```
$ sudo ssh-copy-id -i /home/manal/.ssh/id_rsa.pub manal@hdslave3
```

8- تنزيل منصة هادوب من رابط الموقع الرسمي في جميع العقد الموجودة في العنود:

```
manal@master:~$ wget https://archive.apache.org/dist/hadoop/core/hadoop-2.9.0/hadoop-2.9.0.tar.gz
```

9- تعديل الملف .bashrc. وإضافة متحول بيئة يحوي مسار جافا وأيضاً إضافة مسار هادوب كمتحول إلى البيئة في جميع العقد

```
$ sudo gedit .bashrc
```

```
export HADOOP_PREFIX=/usr/hadoop
export HADOOP_HOME=/usr/hadoop
export HADOOP_MAPRED_HOME=${HADOOP_HOME}
export HADOOP_COMMON_HOME=${HADOOP_HOME}
export HADOOP_HDFS_HOME=${HADOOP_HOME}
export YARN_HOME=${HADOOP_HOME}
export HADOOP_CONF_DIR=${HADOOP_HOME}/etc/hadoop
# Native Path
export HADOOP_COMMON_LIB_NATIVE_DIR=${HADOOP_PREFIX}/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_PREFIX/lib"
# Java path
export JAVA_HOME='/usr/lib/jvm/java-8-openjdk-amd64'
# Add Hadoop bin/ directory to PATH
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_PATH/bin:$HADOOP_HOME/sbin
```

10- إضافة مسار جافا إلى الملف hadoop-env.sh في جميع العقد:

```
export JAVA_HOME='/usr/lib/jvm/java-8-openjdk-amd64'
```

11- تعديل الملف core-site.xml ليصبح على الشكل التالي في جميع العقد:

```
<property>
<name>fs.defaultFS</name>
<value>hdfs://master:9000</value>
</property>
```

12- تعديل الملف hdfs-site.xml ليصبح على الشكل التالي في العقدة الرئيسية:

```
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
<property>
<name>dfs.namenode.name.dir</name>
<value>file:/usr/hadoop/hadoop_data/hdfs/namenode</value>
</property>
```

13- تعديل الملف hdfs-site.xml ليصبح على الشكل التالي في العقدة الأخرى:

```
<property>
<name>dfs.datanode.data.dir</name>
<value>file:/usr/hadoop/hadoop_data/hdfs/datanode</value>
</property>
```

14- تعديل الملف yarn-site.xml ليصبح على الشكل التالي في العقدة الرئيسية والعقد الأخرى:

```

<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
<property>
<name>yarn.nodemanager.auxservices.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
<property>
<name>yarn.resourcemanager.resource-tracker.address</name>
<value>master:8025</value>
</property>
<property>
<name>yarn.resourcemanager.scheduler.address</name>
<value>master:8030</value>
</property>
<property>
<name>yarn.resourcemanager.address</name>
<value>master:8050</value>
</property>
<property>
<name>yarn.nodemanager.vmem-check-enabled</name>
<value>false</value>
</property>

```

15- تعديل الملف mapred-site.xml في العقد الرئيسية والعقد الأخرى ليصبح على الشكل التالي:

```

<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>master:54311</value>
</property>
<property>
<name>mapreduce.jobhistory.address</name>
<value>master:10020</value>
<description>Host and port for Job History Server (default 0.0.0.0:10020)</description>
</property>

```

- 16- تعديل الملف master ووضع اسم العقد الرئيسية داخله وتعديل الملف slaves ووضع اسم العقد الأخرى بداخله.  
 17- نقوم بإنشاء المسار التالي في نظام ملفات هادوب الموزّع على العقد الرئيسية:

```
$ sudo mkdir -p /usr/hadoop/hadoop_data/hdfs/namenode
```

- 18- وكذلك على كل عقد بيانات إنشاء المسار التالي /usr/hadoop/hadoop\_data/hdfs/datanode  
 19- تشغيل عنقود هادوب حيث نقوم بإجراء تهيئة للعقد الرئيسية عند بدء التشغيل الأول ثم نقوم بإجراء:

```
$ /usr/hadoop/bin/hadoop namenode –format
```

```
$ /usr/hadoop/sbin/start-all.sh
```

<b>manal@master:~\$ jps</b>	<b>manal@hdslove1:~\$ jps</b>	<b>manal@hdslove2:~\$ jps</b>	<b>manal@hdslove3:~\$ jps</b>
5667 NameNode	5778 NodeManager	5698 NodeManager	5896 DataNode
6344 Jps	5955 Jps	5558 DataNode	5979 Jps
6059 ResourceManager	5630 DataNode	5871 Jps	5662 NodeManager
5900 SecondaryNameNode			

الشكل (3-5) وكلاع نظام هادوب العاملة على جميع العقد.

### 3-2 تثبيت HiBench على العقد الرئيسية وربطه مع هادوب:[65]

سنستخدم في التجارب المجموعة المرجعية المفتوحة المصدر HiBench والتي تستخدم لتقدير أداء أطر البيانات الضخمة هادوب وسبارك وت تكون من أعباء العمل لمختلف أنظمة البيانات الكبيرة ومولد بيانات الذي يولد البيانات من أحجام مختلفة لأعمال العمل تلك. إن أعباء العمل مجردة من حيث المقارنة ويمكن تكوينها بسهولة لتوليد وتشغيل معايير HiBench و هو متوافق مع جميع التوزيعات الرئيسية لأطر عمل هادوب وسبارك مثل: أباتشي و كلاوديرا و Hortonworks.

تستخدم HiBench نظام Hadoop Ecosystem الذي يتضمن مكتبات وأطرًا للبرامج مثل MapReduce و Storm و Flink و Nutch و Hive و Flink وكذلك يقوم بتشكيل معظم أعمال عمل سبارك.

المقاييس المستخدمة في HiBench هي: وقت التنفيذ لكل حمل عمل، والإنتاجية، وكمية استخدام موارد النظام. في نهاية تشغيل حمل العمل، يتم إلحاقي جميع هذه المقاييس بملف إخراج من أجل تحليلها. كما يوفر HiBench أيضًا إخراجًا يستند إلى الويب يعرض استخدام موارد النظام لكل عبء عمل تم تنفيذه.

إن الإصدار الحالي من HiBench هو 7.0 يحتوي على 19 عبء عمل مصنفة إلى ستة مجالات رئيسية وهي Micro و Streaming و Graph و Web Search و Machine Learning (التعلم الآلي) و SQL و Web Search.

يستخدم HiBench من أجل Micro Benchmarks مولد بيانات قابل للإعداد والذي يستخدم في الغالب Bayesian RandomTextWriter لكتابه نص ثانوي Binary Text مباشرة إلى HDFS. وبالنسبة لـ Web Search Wikipedia page يستخدم HiBench Classification قاعدة بيانات ارتباطات صفحة إلى صفحة الخاصة بويكيبيديا (to-page link database) وملف تفريغ Wikipedia dump على التوالي، بالنسبة لـ KMeans قاموا بتطوير مولد بيانات عشوائي باستخدام التوزيع الإحصائي لتوليد البيانات. يقوم HiBench في الغالب بإنشاءمجموعات بيانات غير مهيكلة تستهدف نظام ملفات هادوب.

#### 1- تنزيل HiBench من الرابط التالي وفك ضغطه في المسار الذي تريده على العقدة الرئيسية:

<https://github.com/intel-hadoop/HiBench> | GitHub, Inc. (US)

2- بناء HiBench package

```
[INFO] hibench-common ..... SUCCESS [01:16 min]
[INFO] HiBench data generation tools ..... SUCCESS [ 58.705 s]
[INFO] sparkbench ..... SUCCESS [ 0.114 s]
[INFO] sparkbench-common ..... SUCCESS [01:29 min]
[INFO] sparkbench-micro-benchmark ..... SUCCESS [ 49.741 s]
[INFO] sparkbench-machine-learning-benchmark ..... SUCCESS [01:18 min]
[INFO] sparkbench-websearch ..... SUCCESS [ 37.704 s]
[INFO] sparkbench-graph ..... SUCCESS [ 55.902 s]
[INFO] sparkbench-sql ..... SUCCESS [ 33.370 s]
[INFO] sparkbench-streaming ..... SUCCESS [ 43.053 s]
[INFO] sparkbench-project-assembly ..... SUCCESS [ 39.777 s]
[INFO] flinkbench ..... SUCCESS [ 0.043 s]
[INFO] flinkbench-streaming ..... SUCCESS [17:42 min]
[INFO] gearpumpbench ..... SUCCESS [ 0.085 s]
[INFO] gearpumpbench-streaming ..... SUCCESS [18:44 min]
[INFO] hadoopbench ..... SUCCESS [ 0.041 s]
[INFO] hadoopbench-sql ..... SUCCESS [ 13.133 s]
```

. HiBench (4-5) بناء

► لبناء معايير هادوب وسبارك معاً:

\$ mvn -Phadoopbench -Psparkbench -Dspark=2.1 -Dscala=2.11 clean package

3- تحديد إصدار Scala: لتحديد إصدار Scala استخدم Scala=xxx. بشكل افتراضي يكون الإصدار (2.10) أو (2.11).

\$ mvn -Dscala=2.10 clean package

4- ربط هادوب مع Hibench:

► المتطلبات: Python 2.6=< bc (من أجل توليد التقارير) وبدء تشغيل HDFS و Yarn في العنقود.

► إنشاء وتعديل الملف conf/hadoop.conf وإضافة قيم الخصائص التالية الموجودة داخل الملف بشكل صحيح:

```
# Hadoop home
hibench.hadoop.home      /usr/hadoop

# The path of hadoop executable
hibench.hadoop.executable    /usr/hadoop/bin/hadoop

# Hadoop configraution directory
hibench.hadoop.configure.dir  /usr/hadoop/etc/hadoop

# The root HDFS path to store HiBench data
hibench.hdfs.master        hdfs://master:9000

# Hadoop release provider. Supported value: apache, cdh5, hdp
hibench.hadoop.release      apache
```

✓ إعداد الملف conf/ hibench.conf وإدراج السطور التالية:

```
hibench.masters.hostnames master
hibench.slaves.hostnames hdslave1 hdslave2 hdslave3
```

✓ كما يمكن التحكم بحجم ملفات الدخل من خلال إعطاء أحد القيم ( tiny, small, large, huge, gigantic ) للخاصية hibench.scale.profile (bigdata) كما يمكن التحكم بعدد المقابلات المختلفة في هادوب للتحكم بدرجة التفرع كما هو موضح في الشكل:

```
hibench.scale.profile          large
# Mapper number in hadoop, partition number in Spark
hibench.default.map.parallelism 4

# Reducer nubmer in hadoop, shuffle partition number in Spark
hibench.default.shuffle.parallelism 2
```

### 5-3 اختيار مجموعة من أعباء الحمل:

نظراً لأننا نركز أساساً على مقارنة الأداء بين هادوب وسبارك، فسيتم استخدام خوارزمية شائعة الاستخدام [1] في كليهما وهي:

1. **WordCount**: يعتبر المثال الكلاسيكي المدعوم من كلا المنصتين هادوب وسبارك اللذان يستخدمان ماب ريديوس للقيام بال مهمة ، يحسب هذا البرنامج عدد مرات ظهور كل كلمة في ملف الدخل المولد بواسطة RandomTextWriter و يعتمد هذا البرنامج على وحدة المعالجة المركزية بشكل مكثف التي تحسب عدد الكلمات فيفع  $17 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8$  ملء الدخل [67].

يتم تنفيذ هذا البرنامج في هادوب على الشكل التالي:

تأخذ دالة المقابلة كدخل لها سطراً واحداً من ملف الدخل وتقسمه إلى كلمات ويقوم بإصدار زوج من (المفتاح/القيمة) للكلمة على شكل (الكلمة ، 1) بعد ذلك تجمع دالة الاختزال عدد مرات ظهور كل كلمة وتتصدر من أجل كل كلمة زوج من (المفتاح / القيمة) على شكل (الكلمة، المجموع) ويوضع ناتج الاختزال النهائية على HDFS [1].

أما في سبارك يتم في البداية إنشاء RDD عن طريق تحميل البيانات من HDFS باستخدام التابع () textFile يتكون كل سطر من الـ RDD من سطر واحد من الملف الأولى، ثم يطبق التحويلات التالية () map() و flatMap() و reduceByKey() بالترتيب والتي يتم تسجيل البيانات الوصفية الخاصة بها بدون تنفيذها حتى يتم استدعاء الاجراء . saveAsText()

➤ تأخذ RDD مكونة من سطور وتحولها إلى RDD مكونة كلمات.

➤ يقوم بتحويل RDD الكلمات إلى RDD مكونة من أزواج (word, 1) كما يطلق عليه أيضاً (المفتاح / قيمة).

➤ لكل مفتاح (كلمة) تخض جميع القيم المقابلة له عن طريق جمع جميع القيم معاً، والآن لدينا RDD من أزواج (كلمة ، <عدد مرات ظهورها>).

**Algorithm 1: Word Count in Hadoop**

```

1: class Mapper<K1, V1, K2, V2>
2:   function map(K1, V1)
3:     List words = V1.splitBy(token);
4:     foreach K2 in words
5:       write(K2, 1);
6:     end for
7:   end function
8: end class

1: class Reducer<K2, V2, K3, V3>
2:   function reduce(K2, Iterable<V2> itor)
3:     sum = 0;
4:     foreach count in itor
5:       sum += count;
6:     end for
7:     write(k3, sum);
8:   end function
9: end class

```

**Algorithm 2: Word Count in Spark**

```

1: class WordCount
2:   function main(String[] args)
3:     file = sparkContext.textFile(filePath);
4:     JavaRDD<String> words = flatMap <- file;
5:     JavaPairRDD<String, Integer> pairs = map <- words;
6:     JavaPairRDD<String, Integer> counts = reduceByKey <- pairs
7:     result = sortByKey <- counts;
8:   end function
9: end class

```

**الشكل(5-5) خوارزمية WordCount في كل من هادوب وسبارك[1].**

### 3-4 معايير قياس الأداء المستخدمة والمدعومة من قبل HiBench :

1. **زمن التنفيذ:** وهو الزمن الفاصل بين بداية تنفيذ العمل (البرنامج) ونهايته مقاساً بالثواني.
2. **الإنتاجية:** وهو مقياس لكمية البيانات التي يمكن للنظام معالجتها خلال واحدة الزمن وهي أيضاً معدل بيانات الدخل خلال زمن التنفيذ ويعبر عنه ب(bytes/second).
3. **زمن الاسترداد من الفشل:** وهو الزيادة في زمن تنفيذ البرنامج عند حدوث فشل، عند اختيار المنصة المتقوقة من ناحية التسامح مع الأعطال سيتم على أساس الازدياد الناتج في زمن التنفيذ وليس زمن التنفيذ الكلي بعد إتمام العمل أثناء حدوث فشل.

### 3-4-1 منهجة البحث المستخدمة في تنفيذ التجارب:

1. في كلا النظمين تم اختبار الاعدادات كعدد المقابلات والمختزلات في هادوب وعدد المنفذين في سبارك وتم انجاز التجارب وفق الاعدادات التي يعطي من أجلها كلا النظمين أفضل أداء ممكن.
2. تم اختيار البرنامج تبعاً للحمل الذي يؤثر به على النظام (WordCount) مستهلك للمعالج).
3. تم اختيار مدير العنقود لسبارك Yarn .
4. سيتم تنفيذ كل برنامج خمس مرات وأخذ متوسط قيم المعاملات وذلك للحصول على أدق النتائج.
5. لحظة حق الفشل سوف يتم تحديدها على أساس نسبة مئوية من متوسط زمن تنفيذ البرنامج بدون فشل (مثلاً): لو كان زمن تنفيذ برنامج عشر دقائق في هادوب وثمان دقائق في سبارك وتم اختيار لحظة حقيقة الفشل بعد مرور 25% من الزمن اللازم لإنجاز تنفيذ البرنامج هذا يعني أن الفشل سيتم حقته بعد مرور دقيقتان ونصف في هادوب ودقيقتين في سبارك.
6. عند اختيار المنصة المتقوقة من ناحية التسامح مع الأعطال سيتم على أساس الازدياد الناتج في زمن التنفيذ وليس زمن التنفيذ الكلي بعد إتمام العمل أثناء حدوث فشل.

**3-5 تشغيل حمل العمل الخاص ببرنامج Hadoop WordCount: ويتم ذلك على مرتين:**

- تشغيل الملف `prepare.sh` الذي يطلق مهمة Hadoop المسؤولة عن توليد بيانات الدخل على HDFS :

`bin/workloads/micro/wordcount/prepare/prepare.sh`

```
patching args=
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/hadoop.conf
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/hibench.conf
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/workloads/micro/wordcount.conf
probe sleep jar: /usr/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.9.0-tests.jar
start HadoopPrepareWordcount bench
hdfs rm -r: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop fs -rm -r -skipTrash hdfs://master:9000/HiBench/Wordcount/Input
Deleted hdfs://master:9000/HiBench/Wordcount/Input
Submit MapReduce Job: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop jar /usr/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.9.0.jar randomtextwriter -D mapreduce.randomtextwriter.totalbytes=1073741824 -D mapreduce.randomtextwriter.bytespermap=268435456 -D mapreduce.job.maps=4 -D mapreduce.job.reduces=2 hdfs://master:9000/HiBench/Wordcount/Input
The job took 45 seconds.
finish HadoopPrepareWordcount bench
manal@master:~$
```

✓ يمكن رؤية الملفات المولدة من التعليمية السابقة عن طريق وضع الرابط التالي في المتصفح:

<http://master:50070/explorer.html#/HiBench/Wordcount/Input>

Group	Size	Last Modified	Replication	Block Size	Name
supergroup	0 B	Oct 08 07:58	3	128 MB	_SUCCESS
supergroup	262.79 MB	Oct 08 07:58	3	128 MB	part-m-00000
supergroup	262.8 MB	Oct 08 07:58	3	128 MB	part-m-00001
supergroup	262.79 MB	Oct 08 07:58	3	128 MB	part-m-00002
supergroup	262.79 MB	Oct 08 07:58	3	128 MB	part-m-00003

**الشكل (5-6) ملفات الدخل لبرنامج Sort و WordCount**

- ✓ أو من خلال التعليمية التالية: `/usr/hadoop/bin/hadoop dfs -ls /HiBench/Wordcount/Input`
- ✓ حيث يتم التحكم بمعنى أحجام ملف الدخل المحددة بالملف السابق كأرقام تدرج بملف الإعداد الخاص بكل حمل عمل على حدا فمثلاً من أجل برنامج wordcount على ملف الإعداد wordcount.conf الموجود في المسار التالي:  
`/HiBench-master/conf/workloads/micro/ wordcount.conf`
- ✓ ويتم أيضاً تحديد المجلد الذي سيوضع فيه ملفات الدخل للبرنامج وملفات الخرج الخاصة به:

<code>hibench.wordcount.tiny.datasize</code>	3200000
<code>hibench.wordcount.small.datasize</code>	32000000
<code>hibench.wordcount.large.datasize</code>	1073741824
<code>hibench.wordcount.huge.datasize</code>	32000000000
<code>hibench.wordcount.gigantic.datasize</code>	320000000000
<code>hibench.wordcount.bigdata.datasize</code>	1600000000000

- ✓ من خلال الرابط `master:8088/cluster` يتم عرض واجهة ال webUI تحوي عن تفاصيل لجميع البرامج التي يتم اطلاقها وتنفيذها ضمن العنقود :

The screenshot shows the Hadoop Web UI interface. At the top, it displays 'All Applications' and 'Browsing HDFS' with a URL 'master:8088/cluster'. The main area features a large 'hadoop' logo. On the left, there's a sidebar with a tree view under 'Cluster' containing 'About', 'Nodes', 'Node Labels', 'Applications' (with sub-options: NEW, NEW\_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED), and 'Scheduler'. Below the sidebar is a 'Tools' button. The central part of the page is titled 'Cluster Metrics' with four tabs: 'Apps Submitted' (6), 'Apps Pending' (0), 'Apps Running' (0), and 'Apps Completed' (6). Below this is 'Cluster Nodes Metrics' with tabs for 'Active Nodes' (3) and 'Decommissioning Nodes' (0). Under 'Scheduler Metrics', it shows 'Scheduler Type' as 'Capacity Scheduler' and 'Scheduling Resource Type' as '[MEMORY]'. A table lists six applications:

ID	User	Name	Application Type	Queue	Application Priority
application_1540548290305_0006	manal	word count	MAPREDUCE	default	0
application_1540548290305_0005	manal	word count	MAPREDUCE	default	0
application_1540548290305_0004	manal	word count	MAPREDUCE	default	0
application_1540548290305_0003	manal	word count	MAPREDUCE	default	0
application_1540548290305_0002	manal	word count	MAPREDUCE	default	0
application_1540548290305_0001	manal	random-text-writer	MAPREDUCE	default	0

At the bottom, it says 'Showing 1 to 6 of 6 entries'.

الشكل (7-5) واجهة الويب الخاصة بتنفيذ هادوب لبرنامج WordCount خمس مرات.

- ✓ تشغيل run.sh لإرسال برنامج هادوب إلى العنود وإطلاق تنفيذه كما يمكن استخدام bin/run\_all.sh لتشغيل جميع أحمال العمل المدرجة في .conf / frameworks.lst و conf / benchmarks.lst.

```

patching args=
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/hadoop.conf
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/hibench.conf
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/workloads/micro/wordcount.conf
probe sleep jar: /usr/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.9.0-tests.jar
start HadoopWordcount bench
hdfs rm -r: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop fs -rm -r -sk
ipTrash hdfs://master:9000/HiBench/Wordcount/Output
Deleted hdfs://master:9000/HiBench/Wordcount/Output
hdfs du -s: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop fs -du -s hdfs://master:9000/HiBench/Wordcount/Input
18/10/26 03:53:54 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Submit MapReduce Job: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop jar
/usr/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.9.0.jar wordcount -D mapreduce.job.maps=4 -D mapreduce.job.reduces=2 -D mapreduce.inputformat.class=org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.outputformat.class=org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat -D mapreduce.job.inputformat.class=org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.job.outputformat.class=org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat hdfs://master:9000/HiBench/Wordcount/Input
Bytes Written=23025
finish HadoopWordcount bench
manal@master:~$ 

```

manal@master:~\$ jps 6272 Jps 3521 JobHistoryServer 5297 SecondaryNameNode 5489 ResourceManager 6226 RunJar 5096 NameNode	manal@hdsSlave3:~\$ jps 5360 YarnChild 5187 MRAppMaster 5364 YarnChild 2741 NodeManager 2585 DataNode 5326 YarnChild 5647 Jps	manal@hdsSlave1:~\$ jps 1168 YarnChild 312 YarnChild 347 Jps 188 YarnChild 1597 DataNode 1255 YarnChild 1744 NodeManager	manal@hdsSlave2:~\$ jps 5889 YarnChild 2837 NodeManager 5848 YarnChild 2680 DataNode 5978 Jps 5917 YarnChild 5790 YarnChild
---------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------

الشكل (8-5) وكلاء نظام هادوب على جميع العقد أثناء تنفيذ برنامج WordCount.

- ✓ من أجل تدوين النتائج الأصح قمنا بإعادة تنفيذ البرنامج WordCount وجميع البرامج التالية في كل من هادوب وسبارك خمس مرات ومن ثم نأخذ متوسط المقاييس بالاعتماد على النتائج في الخمس مرات التي تم تنفيذها.
- ✓ المقاييس التي سوف يتم اعتمادها هي المقاييس التي تقدمها HiBench والتي يتم تخزينها في ملف report/Hibench.report/ الذي يحوي البيانات التالية: نوع البرنامج – تاريخ إطلاق تنفيذه-وقت الإطلاق -حجم بيانات الدخل(بايت) – المدة التي استغرقها التنفيذ بالثواني – الإنتاجية على مستوى العنود (بايت/الثانية).

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
HadoopWordcount	2021-06-02	03:57:05	1102229595	190.021	5800567	1933522
HadoopWordcount	2021-06-02	04:03:51	1102229595	135.970	8106417	2702139
HadoopWordcount	2021-06-02	04:18:00	1102229595	220.045	5009109	1669703
HadoopWordcount	2021-06-02	04:24:33	1102229595	235.897	4672503	1557501
HadoopWordcount	2021-06-02	04:29:46	1102229595	197.353	5585066	1861688

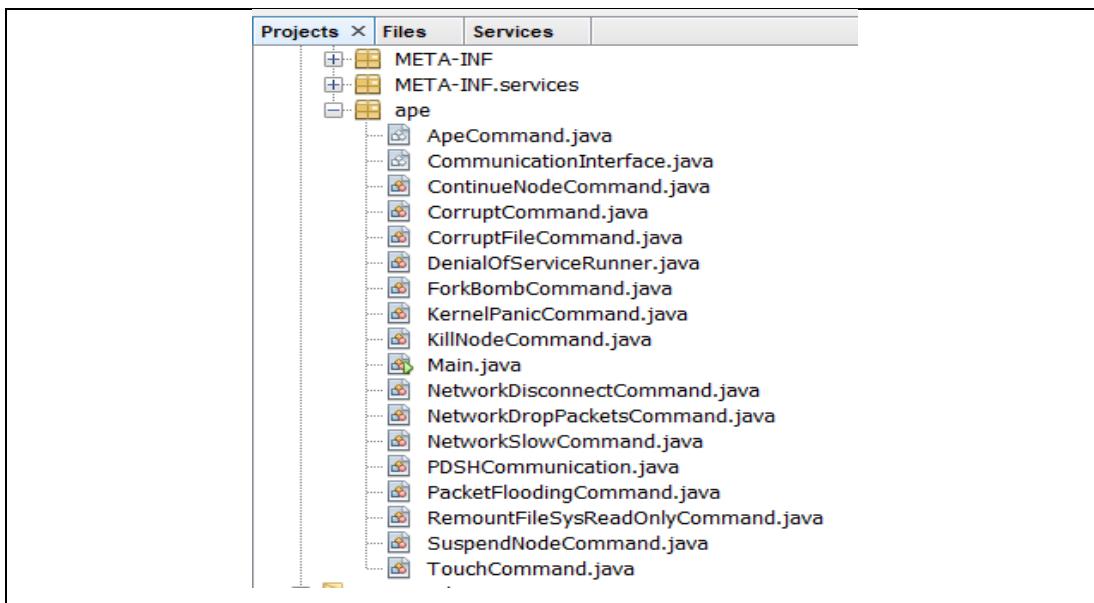
الجدول (1-5) قيم معلمات الأداء لبرنامج هادوب word count بدون فشل.

- ✓ اسم البرنامج HadoopWordcount: حجم البيانات(بايت) = 1 غيغا بايت = 1073741824 بايت، يقسم هذا الحجم إلى أجزاء متساوية ليتم إدخالها إلى المقابلات عدد ال Mapper=4 ، عدد ال Reducer=2 . متوسط زمن التنفيذ =  $\frac{196,0372}{5} = \frac{190,021}{980,186} = \frac{197,353+235,897+220,045+135,970}{5} = 197,353$  بايت/ثانية (3 دقائق و 16 ثانية).
- 1 - الإنتاجية في العنود =  $\frac{5585066+4672503+5009109+8106417+5800567}{5} = 24968409$  بايت / ثانية (22,5 ميغابايت/ثانية).
- 2 - الإنتاجية في العنود =  $\frac{4993681,8}{24968409} = 5585066+4672503+5009109+8106417+5800567$  بايت / ثانية (22,5 ميغابايت/ثانية).

### 6-3-5 اختيار مجموعة من حالات الفشل الممكنة في كلا المنصتين:

يهدف اختبار التسامح مع الفشل إلى العثور على أعطال في تنفيذ أو في مواصفات آليات التسامح مع الفشل، لهذا الغرض يتم تنفيذ النظام في بيئة اختبار مع حقن أعطال معروفة [6] ، عادة ما يتم قياس أداء وظيفة ماب ريديوس من خلال وقت التنفيذ، ويمكن أن يتأثر بشكل كبير بعوامل عديدة مثل الشبكة أو بروتوكول الاتصال وعدد مهام المقابلة وعدد مهام الاختزال ونمط بيانات المرحلية التي يتم خلطها وحجم البيانات المخلوطة [19].

سوف نستخدم الأداة AnarchyApe [70] وهو عبارة عن مشروع مفتوح المصدر أنشأته ياهو Yahoo وطورته لحقن الفشل في عناقيد Hadoop 1.0 عام 2012 لذلك قمنا بتطوير الأداة لحقن الفشل في بيئة YARN 2.0 و Apache Spark، يدعم AnarchyApe ستة عشر اخفاقاً شهيراً ضمن الأنظمة الموزعة كل اخفاق يمثل بتعليميه أو أمر وكل حالة فشل تمثل بصف بلغة الجافا:



. الشكل(9-5) أنواع الفشل المدعومة من قبل الأداة . AnarchyApe

- ✓ ترجمة الأداة **compilation**: تم وضع الأداة على العقدة الرئيسية حيث سيتم ترجمتها وتنفيذ الفشل على العقدة الرئيسية محلياً أو على باقي العقد في العنقود عن بعد باستخدام بروتوكول PDSH [7]:

```
manal@master:/usr/local/bin/src/main/java$ javac -cp .:log4j-1.2.14.jar:commons-cli-1.2.jar ape/*.java
```

- ✓ تشغيل حالات الفشل **Running**: يتم تنفيذ كل حالة فشل عن طريق تعليمية في موجه الأوامر.

مثال: تعليمية **anarchyape** لإنهاء مدير التطبيق الذي يعمل على العقدة1 :hdslave1

```
manal@master:/usr$ java -cp .:log4j-1.2.14.jar:commons-cli-1.2.jar -jar anarchy_ape.jar -R hdslave1 -k mrappmaster [-R, hdslave1, -k, mrappmaster] [ option: R remote :: Run commands remotely ] [ option: k kill-node :: Kills a datanode, nodemanager, resourcemanager, or namenode. ] Mode is remote List of Hosts: hdslave1 Command is kill-node Command Argument: mrappmaster pdsh -Rssh -w hdslave1 '/usr/local/bin/anarchy_ape.jar' -L -k mrappmaster' echo "pdsh -Rssh -w hdslave1 '/usr/local/bin/anarchy_ape.jar -L -k mrappmaster'" > 9f948a17-1b91-4434-9a9d-be39498cc74e.sh && chmod +x 9f948a17-1b91-4434-9a9d-be39498cc74e.sh && rm 9f948a17-1b91-4434-9a9d-be39498cc74e.sh Running Remote Command Succeeded
```

### 7-3-5 تنفيذ وتقييم أداء هادوب عند أعباء الحمل المختلفة مع حقن حالات الفشل في مكونات البيئة:

#### 1-7-3-5 تشغيل حمل العمل الخاص ببرنامج WordCount مع فشل مدير العقدة **Node Manager**

- ✓ لن يتم قتل مدير عقدة عشوائي في العنقود بل سوف نعمل على قتل مدير العقدة التي لجا إليها مدير التطبيق وأطلق حاويات مساعدة عليها ودراسة قدرة هادوب على تجاوز فشل مدير العقدة الذي سيتم حققه بشكل دائم بعد مرور ما يقارب ربع الزمن اللازم لإنجاز التطبيق دون وجود فشل.

قبل حقن الفشل:

```

manal@hdslave3:~$ jps
9921 MRAppMaster
10275 Jps
9492 NodeManager
10069 YarnChild
10040 YarnChild
9352 DataNode
10061 YarnChild
10093 YarnChild
10094 YarnChild
10111 YarnChild
manal@hdslave2:~$ jps
8263 YarnChild
7720 NodeManager
8281 YarnChild
8297 Jps
7583 DataNode
manal@hdslave1:~$ jps
8787 YarnChild
8116 DataNode
8726 YarnChild
8698 YarnChild
8253 NodeManager
8799 Jps

```

✓ بعد حرق الفشل:

```

manal@master:/usr$ java -cp .:log4j-1.2.14.jar:commons-cli-1.2.jar -jar anarchy_ape.jar -R hdslave1 -k NodeManager
[ option: R remote :: Run commands remotely ]
[ option: k kill-node :: Kills a datanode, nodemanager, resourcemanager, or namenode. ]
Mode is remote
List of Hosts:
hdslave1
Command is kill-node
Command Argument: NodeManager
pdsh
-Rssh
-w
hdslave1
'/usr/local/bin/anarchy_ape.jar'
-L
-k
NodeManager'
echo "pdsh -Rssh -w hdslave1 '/usr/local/bin/anarchy_ape.jar -L -k NodeManager'" > 66f22139-8692-4fb9-bfa4-f3283e37bd2e.sh && chmod +x 66f22139-8692-4fb9-bfa4-f3283e37bd2e.sh && ./66f22139-8692-4fb9-bfa4-f3283e37bd2e.sh && rm 66f22139-8692-4fb9-bfa4-f3283e37bd2e.sh
Running Remote Command Succeeded

```

```

manal@hdslave2:~$ jps
8610 Jps
8263 YarnChild
7720 NodeManager
7583 DataNode
manal@hdslave2:~$ jps
7720 NodeManager
8653 Jps
7583 DataNode
manal@hdslave1:~$ jps
8992 Jps
8116 DataNode
8698 YarnChild
8116 DataNode
9033 Jps

```

✓ في الشكل السابق تم إطلاق مدير التطبيق على العقدة hdslave3 والذي بدأ حاويات على العقدتين hdslave1 و hdslave2 التي تم قتل مدير عقدتها بعد مرور ربع الوقت اللازم لانهاء العمل في حالة عدم وجود فشل . إذا فشل مدير العقدة فسوف يتوقف عن إرسال ضربات القلب إلى مدير الموارد وسيلاحظ مدير الموارد توقف مدير العقدة عن إرسال ضربات القلب فيقوم بحذفه من قائمة العقد التي سيتم جدولة الحاويات عليها.

Cluster Nodes Metrics				
Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Standby Nodes
2	0	0	1	0

✓ بالإضافة إلى ذلك يقوم مدير التطبيق بإعادة تشغيل مهام المقابلة التي تم تشغيلها وإكمالها بنجاح على مدير العقدة الفاشلة على مدير عقد آخر وهو هنا hdslave1 ليتم إعادة تشغيلها إذا كانت تتتمى إلى أعمال غير مكتملة نظراً لأن مخرجاتها المرحلية الموجودة على نظام الملفات المحلي لمدير العقدة الفاشلة قد لا يمكن الوصول إليها من قبل مهام الاختزال.

✓ بعد اكتشاف الفشل نلاحظ رغم اكتمال مهام المقابلة على العقدة التي تم قتل مديرها ولكن لم يستطع هادوب جلب نتائجها المرحلية إلى مهام الاختزال مما يؤدي إلى فشل في مرحلة الخلط لذلك قام مدير التطبيق بإعادة حجز حاويات على عقد أخرى وإعادة تنفيذ مهام المقابلة على العقدة الجديدة.

```

fetch failures. Failing the attempt. Last failure reported by
1540584887770_0001_r_000001_0 from host hdslave1
13:34:43 INFO mapreduce.Job: map 88% reduce 29%
13:34:59 INFO mapreduce.Job: map 94% reduce 29%
13:35:04 INFO mapreduce.Job: map 100% reduce 29%
13:35:05 INFO mapreduce.Job: map 100% reduce 100%
13:38:05 INFO mapreduce.Job: Job job_1540584887770_0001 completed
ully

Job Counters Shuffled Maps =16
Failed map tasks=2 Failed Shuffles=8

```

✓ تم تنفيذ التطبيق خمس مرات للحصول على نتائج أدق وكانت النتائج كالتالي:

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
HadoopWordCount	2021-06-03	11:25:10	1102229595	990.553	1112741	370913
HadoopWordCount	2021-06-03	12:11:58	1102229595	716.112	1539816	513062
HadoopWordCount	2021-06-03	12:40:57	1102229595	990.456	1112850	370950
HadoopWordCount	2021-06-03	13:08:45	1102229595	980.785	1123823	374607
HadoopWordCount	2021-06-03	13:38:05	1102229595	895.124	1231370	410456

الجدول(2-5) قيم معاملات الأداء لبرنامج WordCount مع فشل مدير العقدة.

- متوسط زمن التنفيذ =  $5 / 4573,03 = 895,124 + 980,785 + 990,456 + 761,112 + 990,553 = 14,606$  دقيقة و 14,606 ثانية.

=  $5 / 6120600 = 1231370 + 1123823 + 1112850 + 1539816 + 1112741 = 1224120$  بait / ثانية ( ١٦,١ ميغابايت / ثانية).

- زمن تجاوز الفشل: الفاصل الزمني بين لحظة حقن الفشل وبعد تجاوزه وهو مساو إلى الازدياد الناتج في زمن التنفيذ عن حالة عدم وجود فشل:  $196,0372 - 914,606 = 718,5688$  ( ١١ د و ٥٨ ث ).

### 2-7-3-5 تشغيل حمل العمل الخاص ببرنامجه WordCount مع فشل مدير التطبيق MRAppMaster :

- ✓ يرسل مدير التطبيق نبضات دورية إلى مدير الموارد وفي حالة حدوث فشل في مدير التطبيق، سيقوم مدير الموارد باكتشاف الفشل وبعد نسخة جديدة من مدير التطبيق ضمن حاوية جديدة (تم إدارتها بواسطة مدير العقدة).
- ✓ عند التنفيذ العملي للتجربة نلاحظ أنه عند حقن الفشل يتم بشكل فوري اكتشافه على عكس حالة فشل مدير العقدة التي تستغرق وقتاً أطول نوعاً ما حيث يقوم مدير الموارد باختيار عقدة بيانات أخرى ليطلق عليها مدير التطبيق من جديد:

Show 20 entries
Attempt ID
appattempt_1540663317396_0001_000002
appattempt_1540663317396_0001_000001
Show 1 to 2 of 2 entries

✓ وعند التنفيذ نلاحظ أنه إذا كان البرنامج قد أنهى مجموعة من مهام الاختزال عند حقن الفشل فإنه لا يتم إعادة تنفيذ مهمات المقابلة التابعة لها أما إذا لم يكن قد بدء بمهام الاختزال عندها سوف يتم إعادة تنفيذ جميع مهام المقابلة حتى المكتملة منها.

✓ قبل حقن الفشل تم إطلاق مدير التطبيق على العقدة 1:hds1ave1

manal@hds1ave2:~\$ jps 2610 NodeManager 2947 YarnChild 3108 Jps 2918 YarnChild 2903 YarnChild 2967 YarnChild 2921 YarnChild 2955 YarnChild 2973 YarnChild 2941 YarnChild 2431 DataNode	manal@hds1ave1:~\$ jps 2832 DataNode 3217 MRAppMaster 2980 NodeManager 3370 Jps
manal@hds1ave3:~\$ jps 2647 DataNode 3223 Jps 2808 NodeManager	

✓ بعد حقن الفشل تم محاولة إعادة تشغيل مدير التطبيق على العقدة 3:hds1ave3

manal@hds1ave3:~\$ jps 3473 YarnChild 3457 YarnChild 3442 YarnChild 3410 YarnChild 3443 YarnChild 3462 YarnChild 2647 DataNode 2808 NodeManager 3658 Jps 3291 MRAppMaster	manal@hds1ave2:~\$ jps 2610 NodeManager 3534 Jps 2431 DataNode 3471 YarnChild	manal@hds1ave1:~\$ jps 2832 DataNode 3410 Jps 2980 NodeManager
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------	-------------------------------------------------------------------------

✓ نتائج التجارب:

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
HadoopWordcount	2021-06-04	11:11:32	1102229595	360.333	3058919	1019639
HadoopWordcount	2021-06-04	11:25:52	1102229595	348.312	3164489	1045829
HadoopWordcount	2021-06-04	11:57:37	1102229595	370.875	2971970	990656
HadoopWordcount	2021-06-04	12:08:53	1102229595	228.263	4828770	1609590
HadoopWordcount	2021-06-04	12:19:55	1102235027	271.878	4054153	1351384

الجدول(3-5) قيم معاملات الأداء لبرنامج هادوب wordCount مع فشل مدير التطبيق.

- 1 متوسط زمن التنفيذ =  $\frac{315,9322}{5} = 63,1864$  =  $271,878 + 228,263 + 370,875 + 348,312 + 360,333$  ثانية (5 دقائق و 15 ثانية).
- 2 الإنتاجية في العقود =  $\frac{180,78301}{5} = 36,15660,2$  بait / ثانية (٣٦,٤٧٣ ميغابايت / ثانية)
- 3 زمن تجاوز الفشل: الفاصل الزمني بين لحظة حقن الفشل وبدء تجاوزه وهو مساو إلى الازدياد الناتج في زمن التنفيذ عن حالة عدم وجود فشل =  $196,0372 - 315,9322 = 119,895$  (١٦ د و ٥٨ ثانية).

### 3-7-3 تشغيل حمل العمل الخاص ببرنامجه WordCount مع فشل التحطمه :Node Crash Failure

- ✓ قمنا بمحاكاة أداء هادوب عند تحطم عقدة البيانات بشكل كامل من خلال إغلاق الجهاز الذي تعمل عليه العقدة بعد مرور ربع الوقت اللازم لإنجاز العمل بدون وجود فشل لقد قمنا باختيار عقدة البيانات التي قام مدير التطبيق بالاستعانة بها لإكمال التطبيق وليس العقدة التي يعمل عليها مدير التطبيق وبما أن هناك عقدتي حساب ويختلف عدد الحاويات التي تم حجزها على كل منها وبما أن اختيار العقدة عشوائي لذلك يختلف زمن التنفيذ ويزداد كلما كان عدد الحاويات المحجوزة على العقدة المتحطمة أكبر.

- ✓ يختلف فشل تحطم العقدة عن فشل توقف كل من مدير العقدة node manager أو عقدة البيانات الموجودة على عقدة الحساب. إن الفرق بين التحطمم الكامل وتحطم الوكلاء كمدير العقدة أو عقدة البيانات أو مدير التطبيق يمكن بإرسال حزمة TCP reset (RST) من قبل نظام التشغيل فقط عند قتل الوكيل والتي تعتبر بمثابة إشارة مبكرة عن حدوث فشل.

✓ نتائج التجارب:

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
HadoopWordcount	2021-06-05	04:36:37	1102235027	1140.370	966559	322186
HadoopWordcount	2021-06-05	05:34:27	1102235027	1442.113	764319	254773
HadoopWordcount	2021-06-05	11:16:34	1102235027	1130.397	975086	325028
HadoopWordcount	2021-06-05	11:45:39	1102235027	1129.221	976102	325367
HadoopWordcount	2021-06-05	12:47:09	1102235027	1414.358	779318	259772

الجدول(4-5) قيم معاملات الأداء لبرنامج هادوب wordcount مع فشل التحطمم.

- 1 متوسط زمن التنفيذ =  $\frac{1251,2918}{5} = 250,2918$  =  $1414,358 + 1129,221 + 1130,397 + 1442,113 + 1140,370$  دقيقة (٢٠ دقيقة و ٥١ ثانية).
- 2 الإنتاجية في العقود =  $\frac{892276,8}{5} = 178,45556$  بait / ثانية (٨٤٦,٠٠ ميغابايت / ثانية).
- 3 زمن تجاوز الفشل: الفاصل الزمني بين لحظة حقن الفشل وبدء تجاوزه وهو مساو إلى الازدياد الناتج في زمن التنفيذ عن حالة عدم وجود فشل =  $196,0372 - 1251,2918 = 105,2546$  (١٧ د و ٣٥ ثانية).

### 3-8 تثبيت سبارك وربطه مع :HiBench

**1- تنزيل scala على جميع العقد:**

```
manal@master:~$ sudo wget www.scala-lang.org/files/archive/scala-2.11.0.deb
```

```
manal@master:~$ scala -version
Scala code runner version 2.11.0 -- Copyright 2002-2013, LAMP/EPFL
```

**2- تنزيل سبارك على جميع العقد وتنبيهه على المسار** :/usr/spark**Download Apache Spark™**

1. Choose a Spark release: 2.2.3 (Jan 11 2019) ▾
2. Choose a package type: Pre-built for Apache Hadoop 2.7 and later ▾
3. Download Spark: spark-2.2.3-bin-hadoop2.7.tgz

**3- إضافة مسار سبارك إلى ملف bash:**

```
export SPARK_HOME=/usr/spark
export PATH=$PATH:$SPARK_HOME/bin
```

**4- ربط سبارك مع HiBench**

hibench.spark.home	The Spark installation location
hibench.spark.master	The Spark master, i.e. `spark://xxx:7077` , `yarn-client`

**5-3-9 تنفيذ وتقييم أداء سبارك عند أعباء الحمل المختلفة (بدون فشل):**

► لضبط عنقود سبارك فإن هناك منظوريين لتحديد عدد المنفذين وعدد الأنوية والذاكرة المخصصة لكل منفذ [76] وهم:

**1- الحالة الأولى (منفذ واحد لكل نواة على كل عقدة):**

- عدد المنفذين(num-executors) =  $2 * 3 = 6$ .
- عدد الأنوية لكل منفذ(executor-cores) = 1 لكل منفذ.
- ذاكرة كل منفذ(executor-memory) = ذاكرة العقدة / عدد المنفذين عليها =  $2 / 3 = 1.5$  غيغا بايت.

في هذه الحالة لن نتمكن من الاستفادة من تشغيل مهام متعددة في نفس JVM أيضاً، سيتم تكرار المتغيرات المشتركة في كل نواة ولم نترك مساحة ذاكرة كافية لعمليات Hadoop / Yarn daemon.

**2- الحالة الثانية (منفذ واحد على كل عقدة):**

- عدد المنفذين = (num-executors) .3
- عدد الأنوية لكل منفذ = (executor-cores) .2
- ذاكرة كل منفذ = ذاكرة العقدة / عدد المنفذين عليها = 3 غيغا بايت.

في هذه الحالة لم نترك مساحة ذاكرة كافية لعمليات Hadoop / Yarn daemon على الـ HDFS مع 5 مهام لكل منفذ، لذلك من الجيد الحفاظ على عدد النوى لكل منفذ أقل من هذا الرقم وهذا محقق في حالة العنقود الخاص بنا.

- كما هو الحال في هادوب تم اختبار الحالتين واخترنا في التجارب الحالة الثانية التي أعطت أقصر زمن تنفيذ ممكن كما تمت التجارب من خلال الاعتماد على مدير عنقود هادوب **Yarn** ونمط النشر **Client Mode**.

### 5-3-9-1 تشغيل سبارك لحمل العمل الخاص ببرنامج WordCount بدون فشل:

- ✓ يتم تنفيذ برنامج WordCount على شكل مرحلتين مرحلة المقابلة تحتوي 8 مهام مقابلة توزع على المنفذين الثلاثة :

```
Finished task 0.0 in stage 0.0 (TID 0) in 34705 ms on hdslave1 (executor 3) (1/8)
Finished task 2.0 in stage 0.0 (TID 1) in 34704 ms on hdslave1 (executor 3) (2/8)
Finished task 6.0 in stage 0.0 (TID 3) in 35285 ms on hdslave2 (executor 2) (3/8)
Finished task 4.0 in stage 0.0 (TID 2) in 35432 ms on hdslave2 (executor 2) (4/8)
Finished task 5.0 in stage 0.0 (TID 6) in 14918 ms on hdslave1 (executor 3) (5/8)
Finished task 7.0 in stage 0.0 (TID 7) in 15526 ms on hdslave1 (executor 3) (6/8)
Finished task 3.0 in stage 0.0 (TID 5) in 34116 ms on hdslave3 (executor 1) (7/8)
Finished task 1.0 in stage 0.0 (TID 4) in 34267 ms on hdslave3 (executor 1) (8/8)
```

- ✓ ومرحلة الاختزال تحتوي على أربع مهام اختزال:

```
Finished task 2.0 in stage 1.0 (TID 10) in 2453 ms on hdslave3 (executor 1) (1/4)
Finished task 1.0 in stage 1.0 (TID 9) in 2468 ms on hdslave2 (executor 2) (2/4)
Finished task 0.0 in stage 1.0 (TID 8) in 3445 ms on hdslave1 (executor 3) (3/4)
Finished task 3.0 in stage 1.0 (TID 11) in 3440 ms on hdslave1 (executor 3) (4/4)
```

- ✓ اسم البرنامج (**ScalaSparkWordcount**)، حجم البيانات (بايت) = 1073741824 بايت، يقسم هذا الحجم إلى أربعة أجزاء متساوية، عدد المنفذين = 3، نمط النشر: **Client Mode**.

hdslave1	master	hdslave3	hdslave2
2582 DataNode	3699 Master	3316 ExecutorLauncher	2800 NodeManager
3430 CoarseGrainedExecutorBackend	4442 SparkSubmit	3366 CoarseGrainedExecutorBackend	3537 Jps
3462 Jps	3132 ResourceManager	3383 Jps	2675 DataNode
2711 NodeManager	2765 NameNode	3082 Worker	3222 Worker
3149 Worker	4574 Jps	2524 DataNode	3501 CoarseGrainedExecutorBackend
	2974 SecondaryNameNode	2654 NodeManager	

الشكل(5-10) وكلاء سبارك على العقد بعد تشغيل برنامج Wordcount

- ✓ نتائج التجارب:

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
ScalaSparkWordcount	2021-06-12	10:49:14	1102231455	114.211	9650834	3216944
ScalaSparkWordcount	2021-06-12	10:52:02	1102231455	85.571	12880899	4293633
ScalaSparkWordcount	2021-06-12	10:54:41	1102231455	84.116	13103707	4367902
ScalaSparkWordcount	2021-06-12	10:57:37	1102231455	86.831	12693985	4231328
ScalaSparkWordcount	2021-06-12	10:59:41	1102231455	87.544	12590599	4196866

الجدول(5-5) قيم معاملات الأداء لبرنامج سبارك WordCount بدون فشل.

- متوسط زمن التنفيذ =  $\frac{87,544 + 86,831 + 84,116 + 114,211}{4} = 91,654$  ثانية
- (1 دقيقة و32 ثانية).
- الانتاجية في العنقودية =  $12590599 + 12693985 + 13103707 + 12880899 + 9650834$  بآيت/ثانية.
- الانتاجية =  $12184004,8 = \frac{5,60920024}{11.75}$  ميغابايت /ثانية.

**3-5-10 تنفيذ وتقدير أداء سبارك عند أعباء الحمل المختلفة (مع حن فشل):****1-10-3-5 تشغيل برنامج WordCount مع حن فشل المندف**

- ✓ يتم قتل منفذ عشوائي بعد مرور ربع الوقت اللازم لتنفيذ البرنامج بدون فشل:

```
15:53:22 ERROR cluster.YarnScheduler: Lost executor 2 on hdslave1:  
marked as failed: container_1551015838453_0004_01_000003 on host:
```

✓ مما يؤدي إلى فشل المهام المجدولة للتنفيذ عليه:

```
WARN scheduler.TaskSetManager: Lost task 4.0 in stage 0.0  
WARN scheduler.TaskSetManager: Lost task 6.0 in stage 0.0
```

- ✓ ويتم إزالة هذا المنفذ وجدولة المهام الخاصة به على منفذ موجود أو على منفذ جديد آخر يطلق على نفس العقدة أو على عقدة أخرى إذا دعت الحاجة لذلك:

```
INFO scheduler.TaskSetManager: Starting task 6.1 in stage 0.0  
executor 1, partition 6, NODE_LOCAL, 4874 bytes)
```

- ✓ وبعد تسجيل النتائج نلاحظ قدرة سبارك على اكتشاف الفشل في المنفذين وتجاوزه دون التأثير على الأداء:

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
ScalaSparkWordcount	2021-07-25	21:54:28	1102250836	110.137	10007997	3335999
ScalaSparkWordcount	2021-07-25	21:55:47	1102250836	107.272	10275289	3425096
ScalaSparkWordcount	2021-07-25	21:57:32	1102250836	98.289	11214386	3738128
ScalaSparkWordcount	2021-07-25	21:58:44	1102250836	104.635	10534246	3511415
ScalaSparkWordcount	2021-07-25	21:59:53	1102250836	90.534	12174993	4058331

الجدول(5-6) قيم معلمات الأداء لبرنامج سبارك WordCount مع فشل المنفذ.

1- متوسط زمن التنفيذ =  $\frac{5}{5} / 510,867 = 90,534 + 104,635 + 98,289 + 107,272 + 110,137$  ثانية (102,1734 دقيقة و 54 ثانية).

2- الإنتاجية في العنود =  $10534246 + 12174993 + 11214386 + 10275289 + 10007997$  بايت/ثانية (54206911 ميغابايت /ثانية).

3- زمن تجاوز الفشل: 10,5 ثانية = 91,6546 - 102,1734

**2-10-3-5 تشغيل برنامج WordCount مع حن فشل Executor Launcher**

- ✓ يتم قتل الدا Launcher بعد مرور ربع الوقت اللازم لتنفيذ البرنامج بدون فشل ونلاحظ أن قتل الدا Launcher يؤدي إلى فشل جميع المنفذين الذين قام بإطلاقهم وجميع المهام المجدولة عليهم:

```
ERROR cluster.YarnScheduler: Lost executor 3 on hdslave1:  
INFO scheduler.DAGScheduler: Shuffle files lost for executor: 3 (epoch 0)  
  
ERROR cluster.YarnScheduler: Lost executor 1 on hdslave3:  
INFO scheduler.DAGScheduler: Shuffle files lost for executor: 1 (epoch 1)  
  
ERROR cluster.YarnScheduler: Lost executor 2 on hdslave2:  
INFO scheduler.DAGScheduler: Shuffle files lost for executor: 2 (epoch 2)  
  
WARN scheduler.TaskSetManager: Lost task 3.0 in stage 0.0  
WARN scheduler.TaskSetManager: Lost task 1.0 in stage 0.0
```

- ✓ ونلاحظ أن الدا Launcher في سبارك يقابل مدير التطبيق في هادوب حيث يتم اكتشاف الفشل فيه مباشرة وبدء نسخة جديدة منه على عقدة أخرى:

Attempt ID
appattempt_1551007532483_0005_00002
appattempt_1551007532483_0005_00001

- ✓ ويتم إعادة إطلاق Launcher جديد على عقدة جديدة الذي بدوره يعيد إطلاق منفذين جدد (Executor4) وجدولة تنفيذ المراحل الغير مكتملة مع ما تحوي من مهام من جديد:

```
INFO scheduler.TaskSetManager: Starting task 0.1 in stage 0.0
executor 4, partition 0, NODE_LOCAL, 4874 bytes)
INFO scheduler.TaskSetManager: Starting task 2.1 in stage 0.0
executor 4, partition 2, NODE_LOCAL, 4874 bytes)
```

✓ نتائج التجارب:

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
ScalaSparkWordcount	2021-07-13	12:13:19	1102231455	105.639	10433944	3477981
ScalaSparkWordcount	2021-07-13	12:18:18	1102231455	113.715	9692929	3230976
ScalaSparkWordcount	2021-07-13	12:20:39	1102231455	116.492	9461863	3153954
ScalaSparkWordcount	2021-07-13	12:23:09	1102231455	105.432	10454429	3484809
ScalaSparkWordcount	2021-07-13	12:24:43	1102231455	110.898	9939146	3313048

الجدول(7-5) قيم معاملات الأداء لبرنامج سبارك WordCount مع فشل مطلق المنفذين.

- 1- متوسط زمن التنفيذ =  $\frac{110,4352}{5} = 2,176 = 110,898 + 105,432 + 116,492 + 113,715 + 105,639$  ثانية (1 دقيقة و 50 ثانية).
- 2- الإنتاجية في العنقود =  $\frac{5}{49982311} = 9939146 + 10454429 + 9461863 + 9692929 + 10433944 = 9996462.2$  بايت/ثانية (9,435 ميغابايت/ثانية).
- 3- زمن تجاوز الفشل:  $91,6546 - 110,4352 = 18,7$  ثانية.

### 3-10-3-5 تشغيل برنامج WordCount مع حنف فشل التحطط :

- ✓ لمحاكاة فشل التحطط تم إغلاق العقدة العاملة التي تحوي أحد المنفذين لمحاكاة فشل التحطط:

```
WARN spark.HeartbeatReceiver: Removing executor 1 with no recent heartbeats: 121192 ms exceeds timeout 120000 ms
ERROR cluster.YarnScheduler: Lost executor 1 on hdslave3: Executor heartbeat timed out after 121192 ms
WARN scheduler.TaskSetManager: Lost task 3.0 in stage 0.0 (TID 5, hdslave3, executor 1): ExecutorLostFailure (executor after 121192 ms)
WARN scheduler.TaskSetManager: Lost task 1.0 in stage 0.0 (TID 4, hdslave3, executor 1): ExecutorLostFailure (executor after 121192 ms)
INFO scheduler.DAGScheduler: Executor lost: 1 (epoch 0)
INFO scheduler.TaskSetManager: Starting task 1.1 in stage 0.0 (TID 8, hdslave1, executor 2, partition 1, RACK_LOCAL)
INFO storage.BlockManagerMasterEndpoint: Trying to remove executor 1 from BlockManagerMaster.
INFO scheduler.TaskSetManager: Starting task 3.1 in stage 0.0 (TID 9, hdslave2, executor 3, partition 3, RACK_LOCAL)
INFO storage.BlockManagerMasterEndpoint: Removing block manager BlockManagerId(1, hdslave3, 33519, None)
INFO storage.BlockManagerMaster: Removed 1 successfully in removeExecutor
INFO scheduler.DAGScheduler: Shuffle files lost for executor: 1 (epoch 0)
INFO cluster.YarnClientSchedulerBackend: Requesting to kill executor(s) 1
INFO scheduler.DAGScheduler: Host added was in lost list earlier: hdslave3
```

- ✓ لن ترسل العقدة نبضة قلب وبالتالي سيتم اكتشاف الفشل وستفشل المهام المجدولة على هذا المنفذ وسيتم إعادة اسنادها إلى منفذين آخرين:

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
ScalaSparkWordcount	2021-06-15	20:14:42	1102230755	220.311	5003067	1667689
ScalaSparkWordcount	2021-06-15	20:26:52	1102230755	214.888	5129326	1709775
ScalaSparkWordcount	2021-06-15	20:33:35	1102230755	264.759	4163147	1387715
ScalaSparkWordcount	2021-06-15	20:41:25	1102230755	280.846	3924680	1308226
ScalaSparkWordcount	2021-06-15	20:52:08	1102230755	278.533	3957271	1319090

الجدول(7-21) قيم معاملات الأداء لبرنامج سبارك WordCount مع فشل التحطّم.

- 1- متوسط زمن التنفيذ= $\frac{251,8674}{5/1259,337} = \frac{278,533+280,846+264,759+214,888+220,311}{278,533+280,846+264,759+214,888+220,311}$  ثانية (4 دقيقة و 11 ثانية).
- 2- الإنتاجية في العنقود= $\frac{5/2217491}{3957271+3924680+4163147+5129326+5003067} = \frac{35498,2}{3957271+3924680+4163147+5129326+5003067}$  بایت/ثانية (4.226 ميغابايت/ثانية).
- 3- زمن تجاوز الفشل:  $160,2128 = 91,6546 - 251,8674$  ثانية (2 دقيقة و 40 ثانية).



## الفصل السادس

### النتائج والمقررات

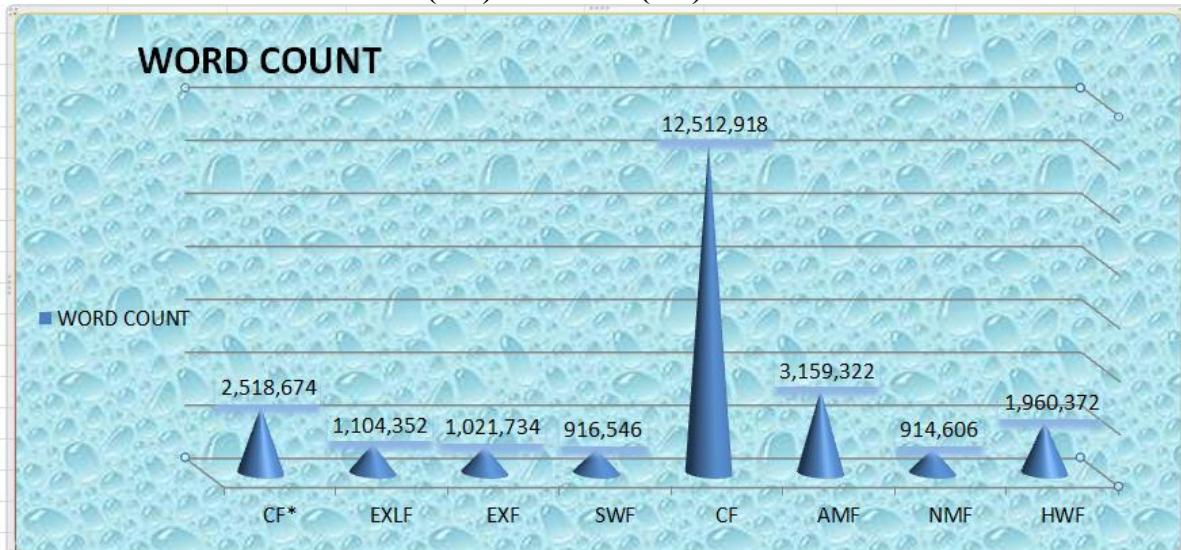
1-8 زمن التنفيذ وزمن تجاوز الفشل (ثانية):

HWF	hadoop without failure
NMF	node manager failure
AMF	application master failure
CF	crash failure
spark wf	spark without failure
EXF	executer failure
EXLF	executer lunch failure
CF*	crash failure

الجدول(1-8) الاختصارات المستخدمة في النتائج.

CF*	EXLF	EXF	SWF	CF	AMF	NMF	HWF	WORD COUNT
2,518,674	1,104,352	1,021,734	916,546	12,512,918	3,159,322	914,606	1,960,372	WORD COUNT

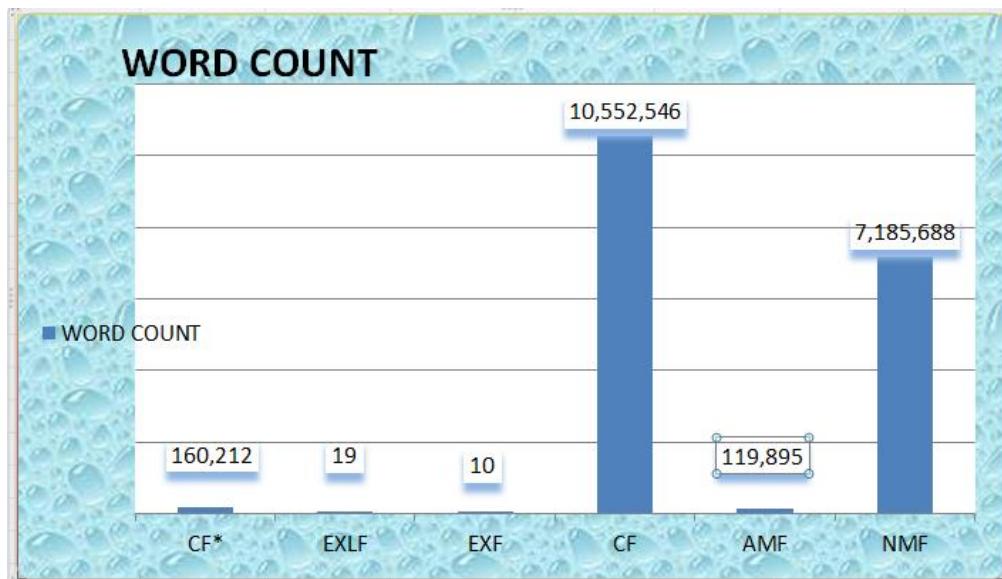
الجدول(2-8) زمن التنفيذ (ثانية)



الشكل(1-8) مخطط زمن التنفيذ (ثانية)

WR F	Spark CF	EX L F	EX F	YC F	Hadoop CF	NM F	AM F	WordCount
3	159.586	18	10	58	1054.444	715.747	117.8716	Sort
0	393.1254	135.277	127.1736	20.798	828.538	140.003	55	pageRank
3	128.4068	37.1192	5	0	881.093	535.223	0	

الجدول(3-8) زمن تجاوز الفشل(ثانية) لجميع التجارب.



الشكل(2-8) مخطط زمن تجاوز الفشل (ثانية) لجميع التجارب.

#### ✓ فشل مدير العقدة:

عند حقن فشل مدير العقدة في برنامج WordCount يتم اكمال تنفيذ جميع مهام المقابلة قبل قدرة مدير الموارد على اكتشاف وجود فشل في مدير العقدة وبعد اكتشافه الفشل، يستغرق وقت كبير في محاولة حصوله على نتائج مهام المقابلة، ويتم رمي فشل في جلب النتائج ، ويُكمل مدير التطبيق العمل من خلال إطلاق حاويات جديدة على عقد آخرى وبدء مهام المقابلة حتى المكتملة منها والتي لم ترسل خرجها إلى مهام الاختزال ولا يعود مدير التطبيق إلى حجز حاويات على العقدة التي فشل مديرها وعندما يتلقى مدير التطبيق إشعاراً بانتهاء المهمة الأخيرة للعمل، فإنه يحاول تحرير الحاويات قبل طباعة رسالة تخبر المستخدم بانتهاء العمل بنجاح وطباعة إحصائيات العمل والعدادات ولذلك تستغرق وقتاً لأن ذلك من مهمة مدير العقدة الفاشل.

#### ✓ فشل مدير التطبيق:

عند التنفيذ العملي للتجربة في برنامج WordCount لاحظنا أنه عند حقن الفشل في مدير التطبيق يتم بشكل فوري اكتشافه من قبل مدير الموارد الذي يقوم بإطلاق نسخة جديدة منه على عقدة أخرى. عند التنفيذ نلاحظ أنه إذا كان البرنامج قد أنهى مجموعة من مهام الاختزال عند حقن الفشل فإنه لا يتم إعادة تنفيذ مهام المقابلة التابعة لها، أما إذا لم يكن قد بدء بمهام الاختزال عندها سوف يتم إعادة تنفيذ جميع مهام المقابلة حتى المكتملة منها، ويتم اكتشاف الفشل وتتجاوزه بسرعة على عكس حالة فشل مدير العقدة التي تستغرق وقتاً أطول نوعاً ما.

#### ✓ فشل تحطم عقدة البيانات في هادوب:

إن الفرق بين التحطّم الكامل وتحطم الوكلاء كمدير البيانات أو عقدة البيانات أو مدير التطبيق، يمكن بإرسال حزمة TCP reset (RST) من قبل نظام التشغيل فقط عند قتل الوكيل ، والتي تعتبر بمثابة إشارة مبكرة عن حدوث الفشل وهذا ما نلاحظه ، حيث يستغرق هادوب ما يقارب العشر دقائق ليكتشف الفشل ويبداً تداركه عند فشل التحطّم سوف يفشل مدير العقدة الخاص بالعقدة المترسبة وجميع المهام المطلقة عليها بعد اكتشاف الفشل ، سيقوم مدير التطبيق بإعادة تنفيذ جميع المهام على عقدة أخرى وسوف يستغرق وقت كبير في محاولة إنهاء الحاويات المحجوزة على العقدة المترسبة ، وهذا ما يؤدي إلى تدهور كبير جداً في زمن التنفيذ الذي يصل إلى سبعة عشر دقيقة في برنامج WordCount

#### ✓ فشل Executor:

يؤدي قتل منفذ عشوائي إلى فشل المهام المجدولة للتنفيذ عليه ، ويتم إزالة هذا المنفذ ويتم إعادة تنفيذ المهام الفاشلة على منفذ آخر أو إطلاق منفذ جديد سواء على نفس العقدة أو عقدة أخرى إذا دعت الحاجة.

في برنامج WordCount تكون المهام الفاشلة مهام مقابلة (stage0.0) لذلك تنفذ بسرعة وزمن تجاوز الفشل في هذا البرنامج سريع جداً.

((يقابل فشل المنفذ في سبارك فشل الـ YarnChild في هادوب ،طبعاً يعتبر عيب في سبارك أنه يتم حجز حاويات ذات حجم ثابت قبل البدء بالتنفيذ ، وبالتالي فشل المنفذ قد يؤدي إلى فشل مهمتين معاً على عكس نظيره هادوب ، الذي يطلق حاوية بشكل ديناميكي تتناسب مع المهمة العاملة عليها وفشل الحاوية يؤدي إلى فشل مهمة واحدة فقط)).

### ✓ فشل Launcher :Executor Launcher

إن قتل Launcher يؤدي إلى فشل جميع المنفذين الذين قام بإطلاقهم وجميع المهام المجدولة عليهم، حيث يتم اكتشاف الفشل فيه مباشرة كما في هادوب ويتم بدء Launcher جديد على عقدة جديدة، الذي بدوره يعيد إطلاق منفذين جدد وجدولة تنفيذ المراحل الغير مكتملة مع ما تحوي من مهام من جديد.

((يقابل مطلق المنفذين في سبارك مدير التطبيق في هادوب، وإذا ما جئنا إلى المقارنة في تجاوز فشهه في كليهما فإننا نلاحظ أنه، في برنامج WordCount يتتفوق سبارك على هادوب في تجاوز فشهه .))

### ✓ فشل التحطط في سبارك:

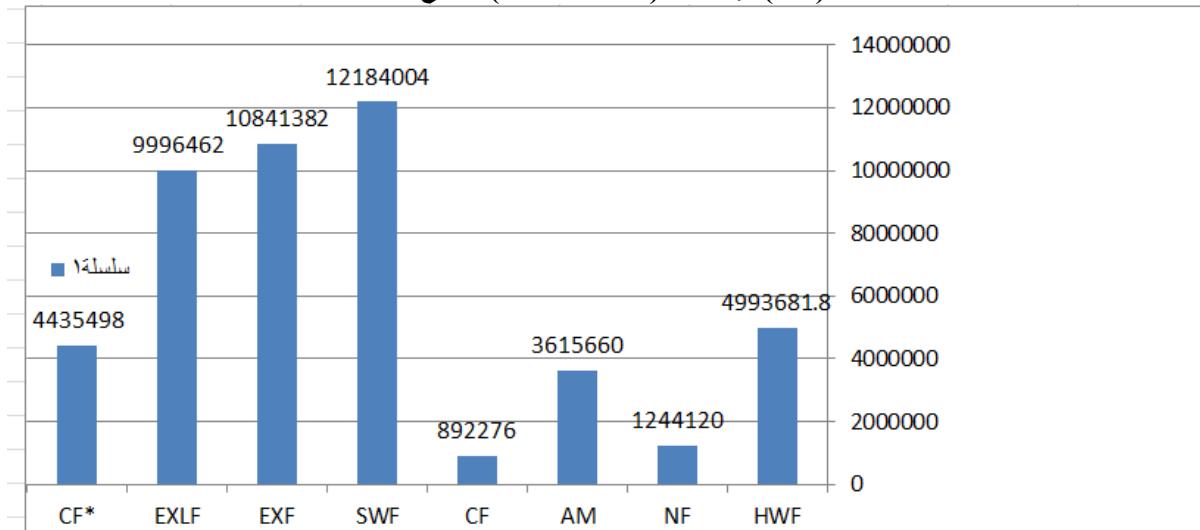
لن ترسل العقدة نبضة قلب وبالتالي سيتم اكتشاف الفشل وستفشل المهام المجدولة على هذا المنفذ وسيتم إعادة اسنادها إلى منفذين آخرين.

((نلاحظ أن سبارك أسرع في تجاوزه بسبب أن عدد الوكلاء العاملين على العقدة ثابت أما هادوب فالعقدة المتحطمها تحوي مدير العقدة وعدد غير محدد من المهام العاملة التي يؤثر قتالها على زمن التنفيذ)).

### 2-8 الإنتاجية (ميغا بايت/ثانية):

CF*	EXLF	EXF	SWF	CF	AM	NF	HWF	
4435498	9996462	10841382	12184004	892276	3615660	1244120	4993682	WORD COUNT

الجدول(4-8) الإنتاجية (ميغا بايت/ثانية) لجميع التجارب.



الشكل(3-8) مخطط الإنتاجية (ميغا بايت/ثانية) لجميع التجارب.

نلاحظ أنه كلما ازداد زمن التنفيذ كلما نقصت إنتاجية النظام وبالتالي، وبما أن الازدياد في التنفيذ في حالة فشل التحطط أكبر من الحالات الأخرى لذلك نجد تدهور كبير في إنتاجية النظام عند وقوعه في كلا النظامين، بينما يعتبر فشل مدير التطبيق في هادوب والمنفذ في سبارك الأقل تأثيراً على الإنتاجية .

### 6-8 النتائج والتوصيات:

نلاحظ أنه من ناحية زمن تجاوز كلا النظامين للأنواع الفشل المحقونة فإن أكثر فشل يؤثر على كلا النظامين هو فشل تحطم عقدة الحساب بما عليها من وكلاء بشكل كامل، إلا أن هادوب يتأثر بشكل أكبر بكثير من سبارك بفشل التحطمم حيث يستغرق ما يقارب **السبعة عشر دقيقة** في برنامج WordCount لتجاوزه، ثم يأتي بعده فشل مدير العقدة إذ يصل زمن تجاوز فشه في برنامج WordCount إلى **اثنا عشر دقيقة** والازدياد في الزمن الحالى أغلبه في محاولة مدير التطبيق تحرير الحاويات على العقدة التي فشل مديرها، بينما يعتبر الفشل في مدير التطبيق الأقل تأثيراً.

يعتبر سبارك في كل من برنامج WordCount متسامح جداً مع جميع أنواع الفشل ماعدا فشل التحطمم ، ورغم ذلك يتفوق على هادوب في تجاوز هذا الفشل، ثم يأتي بعده فشل Launcher بينما يعتبر فشل المُنفذ الأقل تأثيراً .



## المراجع:

- [1] Pan, Shengti, The Performance Comparison of Hadoop and Spark (2016). Culminating Projects in Computer Science and Information Technology .
- [2] MapReduce Tutorial, 2015, <http://hadoop.apache.org/>.
- [3] T. White," Hadoop: The Definitive Guide (Fourth edition)". Sebastopol, CA: O'Reilly Media, 2015.
- [4] Bessani, A. N., Cogo, V. V., Correia, M., Costa, P., Pasin, M., Silva, F., ... Sopena, J. (2010). Making Hadoop MapReduce Byzantine Fault-Tolerant. Proc. of the DSN - Intl. Conf. on Dependable Systems and Networks, (February 2016), 1–2.
- [5] Costa, P., Pasin, M., Bessani, A. N., & Correia, M. P. (2013). On the performance of byzantine fault-tolerant mapreduce. IEEE Transactions on Dependable and Secure Computing, 10(5), 301–313.
- [6] Marynowski, J. E., Santin, A. O., & Pimentel, A. R. (2015). Method for testing the fault tolerance of MapReduce frameworks. Computer Networks, 86, 1–13.
- [7] Faghri, F., Overholt, M., Campbell, R. H., & Sanders, W. H. (2008)." Failure Scenario as a Service ( FSaaS ) for Hadoop Clusters".
- [8] Troubitsyna, Elena A., "Faults, errors, failures," [Online]. Available: <http://users.abo.fi/etroubit/SWS13Lecture2.pdf>.
- [9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder,K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.
- [10] Wang, H., Chen, H., Du, Z., & Hu, F. (2016). BeTL: MapReduce Checkpoint Tactics Beneath the Task Level. IEEE Transactions on Services Computing, 9(1), 84–95.
- [11] Sangroya, A., Serrano, D., Bouchenak, S., Sangroya, A., Serrano, D., Bouchenak, S., & Dependability, B. (2012). Benchmarking Dependability of MapReduce Systems. To cite this version HAL Id: hal-01472165.
- [12] Hao, Z., & Alnawasreh, K. (2016). Distributed Systems verification using fault injection approach.
- [13] Tinetti Fernando, G. Distributed systems: principles and paradigms (2nd edition): Andrew s. tanenbaum, maarten van steen pearson education, inc.,2007 isbn: 0-13-239227-5. Journal of Computer Science and Technology 11, 2 (2011), 115–116.
- [14] Kashkouli, A., & Soleimani, B. (2017). Investigating Hadoop Architecture and Fault Tolerance in Map- Reduce, 17(6), 81–87.
- [15] Ch. Lam, Hadoop in action. Printed in the United States of America, 2011.
- [16] Bilal, K., Khalid, O., Malik, S. U. R., Khan, M. U. S., Khan, S. U., & Zomaya, A. Y. (2016). Fault Tolerance in the Cloud. Encyclopedia of Cloud Computing, (ITProPortal), 291–300.

- [17] Costa, P. A. R. S., Bai, X., Ramos, F. M. V., & Correia, M. (2016). Medusa: An Efficient Cloud Fault-Tolerant MapReduce. Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGGrid 2016, 443–452.
- [18] Evans, J. (n.d.). Fault Tolerance in Hadoop for Work Migration. Salsahpc.Indiana.Edu, 3–7.
- [19] Lu, X., Wasi-ur-Rahman, M., Islam, N. S., & Panda, D. K. (2014). A Micro-benchmark Suite for Evaluating Hadoop MapReduce on High-Performance Networks. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8585, 32–42.
- [20] <https://github.com/david78k/anarchyape>
- [21] Samadi, Y., Zbakh, M., & Tadonki, C. (2017). Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. Concurrency Computation, (October), 1–13.
- [22] Mavridis, I., & Karatza, E. (2015). Log File Analysis in Cloud with Apache Hadoop and Apache Spark. Proceedings of the Second International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2015), 51–62.
- [23] Liu, L. (2015). Performance comparison by running benchmarks on Hadoop, Spark, and HAMR. Retrieved from <https://dspace.udel.edu/handle/19716/17628>
- [24] Samadi, Y., Zbakh, M., & Tadonki, C. (2016). Comparative study between Hadoop and Spark based on Hibench benchmarks.
- [25] Huang, S., Huang, J., Dai, J., Xie, T., & Huang, B. (2011). The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. Lecture Notes in Business Information Processing, 74 LNBP, 209–228.
- [26] Xu, G., Xu, F., & Ma, H. (2012). Deploying and Researching Hadoop in Virtual Machines, 2(August), 395–399.
- [27] Reyes-Ortiz, J. L., Oneto, L., & Anguita, D. (2015). Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. Procedia Computer Science, 53(1), 121–130.
- [28] Kang S, Yeon S, Myung K. Performance comparison of OpenMP, MPI, and MapReduce in practical problems. Adv Multimedia. 2015;2015:9
- [29] Aaron D, Andrew O. Optimizing shuffle performance in Spark. Berkeley-Department of Electrical Engineering and Computer Sciences, California Technical Report.
- [30] Gu L, Li H. Memory or time performance evaluation for iterative operation on Hadoop and Spark. In: IEEE 10th International Conference on High Performance Computing and Communications; 2013; Zhangjiajie.721-727.
- [31] Veiga, J., Exp, R. R., Pardo, C., Taboada, G. L., & Touri, J. (n.d.). Performance Evaluation of Big Data Frameworks for Large-Scale Data Analytics.
- [32] P. Jakovits and S. N. Srivastava, “Evaluating MapReduce frameworks for iterative scientific computing applications,” in Proc. of the International Conference on High Performance Computing & Simulation (HPCS’14), Bologna, Italy, 2014, pp. 226–233.
- [33] Hadoop vs Apache Spark. (n.d.).

[35] Ahmed, H., Ismail, M. A., Hyder, M. F., Sheraz, S. M., & Fouq, N. (2016). Performance Comparison of Spark Clusters Configured Conventionally and a Cloud Service. *Procedia Computer Science*, 82(March), 99–106.

[36] Dinu, F., & Ng, T. S. E. (2012). Understanding the effects and implications of compute node related failures in hadoop. *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing - HPDC '12*, 187.

