การเพิ่มประสิทธิภาพการเข้าถึงไฟล์ขนาดเล็กสำหรับฮาดูปอาร์ไคฟส์

นางสาวจตุพร วรพงศ์กิติพันธ์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2556

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

บทคัดย่อและแฟ้มข้อมูลฉบับเต็มของวิทยานิพนธ์ตั้งแต่ปีการศึกษา 2554 ที่ให้บริการในคลังปัญญาจุฬาฯ (CUIR)
เป็นแฟ้มข้อมูลของนิสิตเจ้าของวิทยานิพนธ์ ที่ส่งผ่านทางบัณฑิตวิทยาลัย
The abstract and full text of theses from the academic year 2011 in Chulalongkorn University Intellectual Repository (CUIR)
are the thesis authors' files submitted through the University Graduate School.

PERFORMANCE IMPROVEMENT OF SMALL-FILE ACCESS FOR HADOOP ARCHIVE

Miss Chatuporn Vorapongkitipun

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering Program in Computer Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2013
Copyright of Chulalongkorn University

หัวข้อวิทยานิพนธ์	การเพิ่มประสิทธิภาพการเข้าถึงไฟล์ขนาดเล็กสาหรับฮาดูปอาร์ไคฟส์
โดย	นางสาวจตุพร วรพงศ์กิติพันธ์
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก	ผู้ช่วยศาสตราจารย์ ดร.ณัฐวุฒิ หนูไพโรจน์

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

....คณบดีคณะวิศวกรรมศาสตร์

(ศาสตราจารย์ ดร.บัณฑิต เอื้ออาภรณ์)

คณะกรรมการสอบวิทยานิพนธ์

....ประธานกรรมการ

(ผู้ช่วยศาสตราจารย์ ดร.เกริก ภิรมย์โสภา)

....อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก

(ผู้ช่วยศาสตราจารย์ ดร.ณัฐวุฒิ หนูไพโรจน์)

....กรรมการ

(ผู้ช่วยศาสตราจารย์ ดร.วีระ เหมืองสิน)

....กรรมการภายนอกมหาวิทยาลัย

(ผู้ช่วยศาสตราจารย์ ดร.ภุชงค์ อุทโยภาศ)

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

จตุพร วรพงศ์กิติพันธ์ : การเพิ่มประสิทธิภาพการเข้าถึงไฟล์ขนาดเล็กสาหรับฮาดูปอาร์ไคฟส์. (PERFORMANCE IMPROVEMENT OF SMALL-FILE ACCESS FOR HADOOP ARCHIVE) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ผศ. ดร.ณัฐวุฒิ หนูไพโรจน์, 80 หน้า.

Hadoop Distributed File System หรือ HDFS เป็นระบบ open source ที่ถูกออกแบบมาเพื่อทำงานบน commodity hardware และเหมาะสำหรับการทำงานกับข้อมูลที่มีขนาดใหญ่ (terabytes) โดยมีโครงสร้างในการทำงานเป็นแบบ master-slaves ซึ่งจะมี NameNode ทำหน้าที่เป็น master จำนวน 1 ตัว ที่คอยทำหน้าที่ในการจัดการกับ metadata ต่างๆของ slaves ต่างๆที่อยู่ภายในระบบ ซึ่งทำให้ NameNode เกิดปัญหาที่เรียกว่าคอขวด โดยเฉพาะอย่างยิ่งเมื่อต้องคอยรองรับการทำงานของไฟล์ขนาดเล็กจำนวนมาก ทั้งนี้เพราะ NameNode จัดเก็บ metadata ทั้งหมดของ HDFS เอาไว้ใน main memory ซึ่งทำให้การใช้งาน memory ไม่มีประสิทธิภาพ เมื่อมีไฟล์ขนาดเล็กจำนวนมาก

จากปัญหาข้างต้น ในงานวิจัยนี้จึงนำเสนอกลไกในการจัดการกับ memory ให้มีความเหมาะสมและเพิ่มประสิทธิภาพในการเข้าถึงไฟล์ขนาดบน HDFS ให้มีประสิทธิภาพที่ดีมากยิ่งขึ้น โดยนำหลักการในการทำงานของ Hadoop Archive หรือ HAR มาใช้เป็นพื้นฐานในการวิจัย โดยที่งานวิจัยนี้จะนำเสนอ Hadoop Archive ในรูปแบบใหม่ที่เรียกว่า New Hadoop Archive (NHAR) ซึ่งเป็นการปรับปรุงโครงสร้างการทำงานของ HAR ขึ้นมาใหม่เพื่อให้มีประสิทธิภาพในการเข้าถึงที่ดีมากยิ่งขึ้น นอกเหนือจากนี้ ในงานวิจัยนี้ยังเพิ่มความสามารถในการทำงานของ HAR โดยการปรับปรุงโครงสร้างการทำงานของ HAR ให้สามารถเพิ่มไฟล์ลงไปไฟล์ archive ที่มีอยู่แล้ว ซึ่งผลลัพธ์ที่ได้จากการทดลองแสดงให้เห็นว่า วิธีที่นำเสนอสามารถเพิ่มประสิทธิภาพในการเข้าถึงไฟล์ข้อมูลขนาดเล็กได้มากถึง 85.47% เมื่อทำการเปรียบเทียบกับการเข้าถึงไฟล์ขนาดเล็กของ HAR

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	....
สาขาวิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก	....
ปีการศึกษา	2556		

5470136021 : MAJOR COMPUTER ENGINEERING
KEYWORDS: HADOOP / HDFS / SMALL FILES IN HADOOP / HADOOP ARCHIVE / HAR / IMPROVE PERFORMANCE

CHATUPORN VORAPONGKITIPUN: PERFORMANCE IMPROVEMENT OF SMALL-FILE ACCESS FOR HADOOP ARCHIVE. ADVISOR: ASST. PROF. NATAWUT NUPAIROJ, Ph.D., 80 pp.

The Hadoop Distributed File System or HDFS is an open source system which is designed to run on commodity hardware and is suitable for applications that have large data sets (terabytes). As HDFS architecture bases on master-slaves architecture. There is one NameNode that serves as master which handle metadata management for multiple slaves, NameNode often becomes bottleneck, especially when handling large number of small files. Since, NameNode stores the entire metadata of HDFS in its main memory. With too many small files, and the memory usage can be inefficient.

In our approach, we propose a mechanism for improve the memory utilization for metadata and enhance the efficiency of accessing small files in HDFS based on Hadoop Archive or HAR, called New Hadoop Archive (NHAR) which re-design the architecture of HAR to improve the efficiency of accessing small files. In addition, we also extend HAR capabilities to allow additional files to be inserted into the existing archive files. Our experiment results show that our approach can to improve the access efficiencies of small files drastically as it outperforms HAR up to 85.47%.

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

Department:	Computer Engineering	Student's Signature ....
Field of Study:	Computer Engineering	Advisor's Signature ....
Academic Year:	2013	

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงได้ด้วยความกรุณาอย่างสูงจากผู้ช่วยศาสตราจารย์ ดร.ณัฐวุฒิ หนูไพโรจน์ อาจารย์ที่ปรึกษาวิทยานิพนธ์ที่กรุณาสละเวลาช่วยเหลือในการให้ความรู้ คำปรึกษา คำแนะนำ รวมทั้งให้แนวคิดที่เป็นประโยชน์ในการทำวิจัย และช่วยชี้แนะการแก้ไขปัญหาต่างๆ ที่เกิดขึ้นระหว่างการทำวิจัย ขอขอบพระคุณอาจารย์มา ณ โอกาสนี้ และขอขอบพระคุณคณาจารย์ทุกท่านที่แนะนำสั่งสอนและให้ความรู้แก่ข้าพเจ้าตลอดระยะเวลาการศึกษา

ขอกราบขอบพระคุณคณะกรรมการสอบวิทยานิพนธ์ทุกท่านเป็นอย่างสูง ผู้ช่วยศาสตราจารย์ ดร.เกริก ภิรมย์โสภา ประธานคณะกรรมการสอบวิทยานิพนธ์ ผู้ช่วยศาสตราจารย์ ดร.วีระ เหมืองสิน กรรมการสอบวิทยานิพนธ์ และ ผู้ช่วยศาสตราจารย์ ดร.ภุชงค์ อุทโยภาศ กรรมการสอบวิทยานิพนธ์ ที่ให้ความกรุณาตรวจสอบและให้ข้อเสนอแนะอันเป็นประโยชน์ในการปรับปรุงแก้ไขวิทยานิพนธ์นี้ให้มีความถูกต้องและสมบูรณ์มากยิ่งขึ้น รวมทั้งคณาจารย์ภาควิชาวิศวกรรมคอมพิวเตอร์ทุกท่านที่ได้ให้ความรู้ และให้การอบรมสั่งสอน

ขอกราบขอบพระคุณ นายพิเชษฐ์ วรพงศ์กิติพันธ์ , นางวนิดา วรพงศ์กิติพันธ์ รวมถึงขอขอบคุณนางสาวฐิติรัตน์ วรพงศ์กิติพันธ์ บิดามารดาและพี่สาวของผู้วิจัยที่ให้การสนับสนุน เอาใจใส่ คอยห่วงใย และให้กำลังใจแก่ผู้วิจัยเสมอมา

ขอขอบคุณนายปัญญา กิตติพิพัฒนถาวร และนายนฤบดี สารสุวรรณ ที่คอยให้คำปรึกษา ให้กำลังใจร่วมแบ่งปันข้อคิดและแลกเปลี่ยนความรู้ในระหว่างการทำวิจัยมาโดยตลอด

นอกจากนี้ ผู้วิจัยขอขอบคุณทุกท่านที่มีส่วนช่วยเหลือจนทำให้วิทยานิพนธ์ฉบับนี้สำเร็จเรียบร้อยลงได้ด้วยดีทุกประการ

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

สารบัญ

หน้า

บทคัดย่อภาษาไทย....................ง
บทคัดย่อภาษาอังกฤษ.................... จ
กิตติกรรมประกาศ.................... ฉ
สารบัญ.................... ช
สารบัญตาราง.................... ญ
สารบัญภาพ .... ฎ
บทที่ 1 บทนำ 13
1.1 ความสำคัญและที่มาของปัญหา 13
1.2 วัตถุประสงค์ 15
1.3 ขอบเขตของงานวิจัย.................... 15
1.4 ประโยชน์ที่คาดว่าจะได้รับ.................... 15
1.5 แผนการดำเนินการวิจัย 16
1.5.1 ศึกษางานวิจัยที่เกี่ยวข้อง 16
1.5.2 ศึกษากลไกในการทำ index file.................... 16
1.5.3 ขั้นตอนออกแบบและทำการสร้าง Framework การทำงานของระบบ.................... 16
1.5.4 ขั้นตอนการทดลองเพื่อทดสอบความสามารถ 16
1.5.5 ขั้นตอนการสรุปผลการทดลองและจัดทำวิทยานิพนธ์ 16
1.6 ผลงานตีพิมพ์.................... 16
บทที่ 2 ทฤษฎีและงานวิจัยที่เกี่ยวข้อง 17
2.1 Hadoop Distributed File System (HDFS) [3, 15].................... 17
2.1.1 สถาปัตยกรรมของ HDFS 17
2.1.1.1 NameNode.................... 18
2.1.1.2 DataNode.................... 19
2.1.2 การเขียนไฟล์ของ HDFS.................... 20
2.1.2.1 การอ่านไฟล์ 20
2.1.2.2 การเขียนไฟล์.................... 21

หน้า

2.1.3 Replica Placement 21
2.2 Hadoop Archive (HAR) [13] 22
2.2.1 The Data Model: har format 22
2.2.2 File system Interface: transparent access 23
2.2.3 The Archiving Tool: MapReduce program สำหรับการสร้าง HAR 24
2.2.4 การอ่านไฟล์จาก HAR File 26
2.3 MapReduce [19] 28
2.3.1 Implementation 28
2.3.2 Example 30
2.3.3 การประยุกต์การใช้งาน MapReduce 31
2.4 Big O (สัญกรณ์โอใหญ่)[20] 32
2.5 ไฟล์ขนาดเล็กบน Hadoop 34
2.6 งานวิจัยที่เกี่ยวข้อง 35
2.6.1 Minimize NameNode 35
2.6.1.1 Local cache 35
2.6.1.2 Specific Data Type 36
2.6.1.3 Sequence File 36
2.6.2 Multiple NameNode 36
2.6.2.1 HDFS Federation 37
2.6.2.2 HAR file 37
บทที่ 3 แนวคิดและวิธีดำเนินงานวิจัย 38
3.1 ปัญหาไฟล์ขนาดเล็กบน Hadoop 38
3.1.1 ข้อจำกัดของพื้นที่จัดเก็บบน NameNode 38
3.1.2 ปัญหาคอขวด (Bottleneck) 39
3.1.3 ปัญหาการเข้าถึงไฟล์ภายใน HAR 40
3.2 หลักการทำงานของระบบ NHAR 41
3.2.1 นิยามของตัวแปรในระบบ NHAR 42

หน้า

3.2.2 ขั้นตอนการสร้าง Archive File ของระบบ NHAR 42
3.2.3 ขั้นตอนการเข้าถึงไฟล์ใน Archive File ของระบบ NHAR 44
3.2.4 ขั้นตอนการเพิ่มไฟล์ใน Archive File ของระบบ NHAR 45
3.3 ตัวอย่างการทำงานของระบบ 48
3.3.1 ตัวอย่างการเพิ่มไฟล์เข้าไปภายใน NHAR file 48
3.4 เปรียบเทียบการเพิ่มไฟล์เข้ายัง HDFS และ NHAR 51
บทที่ 4 การพัฒนาเครื่องมือและการทดลอง 52
4.1 สภาพแวดล้อมที่ใช้ในการทดลอง 52
4.2 การพัฒนาเครื่องมือ 53
4.2.1 ตัวอย่างโครงสร้างของ NHAR 53
4.3 การทดลอง 55
4.3.1 รูปแบบการทดลอง 57
4.3.2 การทดสอบประสิทธิภาพการเข้าถึงไฟล์ขนาดเล็ก 57
4.3.3 การทดสอบประสิทธิภาพในการสร้าง NHAR 59
4.3.4 การทดสอบประสิทธิภาพในการเพิ่มไฟล์ของ NHAR 60
4.3.5 การทดสอบการใช้งานพื้นที่เก็บ (Memory usage) บน NameNode 61
บทที่ 5 บทสรุปและแนวทางในการพัฒนาต่อ 64
5.1 บทสรุป 64
5.2 ข้อจำกัด 65
5.3 แนวทางในการพัฒนาต่อ 65
รายการอ้างอิง 66
ประวัติผู้เขียนวิทยานิพนธ์ 80

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

สารบัญตาราง

หน้า

ตารางที่ 1 แสดงตัวอย่างของเวลาที่ใช้ในการ run ของ algorithm ในแต่ละคลาส 33
ตารางที่ 2 แสดงขนาด metadata ของ scientific application 38

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

สารบัญภาพ

หน้า

ภาพที่ 2.1 แสดงสถาปัตยกรรมของ HDFS 18
ภาพที่ 2.2 แสดงการส่ง heartbeat ระหว่าง NameNode และ DataNode 19
ภาพที่ 2.3 แสดงการอ่านและเขียนไฟล์ของ HDFS 20
ภาพที่ 2.4 แสดงการทำ Archive ไฟล์ขนาดเล็กด้วย HAR file 22
ภาพที่ 2.5 แสดง HAR File Layout 24
ภาพที่ 2.6 Hadoop Archive - Map Task 25
ภาพที่ 2.7 Hadoop Archive - Reduce Task 26
ภาพที่ 2.8 Read file from HAR 27
ภาพที่ 2.9 แสดงกระบวนการทำงานของ MapReduce 29
ภาพที่ 2.10 แสดง order of growth แต่ละ class ของ Big O 34
ภาพที่ 2.11 แสดงวิธีในการแก้ปัญหาไฟล์ขนาดเล็กบน Hadoop 35
ภาพที่ 3.1 กราฟแสดงพื้นที่จัดเก็บ metadata บน NameNode 39
ภาพที่ 3.2 แสดงการเชื่อมต่อไปยัง HDFS 40
ภาพที่ 3.3 แสดงประสิทธิภาพการเข้าถึงไฟล์ของ HAR 41
ภาพที่ 3.4 แสดงโครงสร้างการทำ archive file ของ NHAR 43
ภาพที่ 3.5 แสดงเทคนิคในการทำ archive file ของ NHAR 43
ภาพที่ 3.6 แสดงการเข้าถึงไฟล์ของ NHAR 44
ภาพที่ 3.7 แสดงการเพิ่มไฟล์เข้าไปยัง HAR file ที่มีอยู่แล้วของ HAR 45
ภาพที่ 3.8 แสดงการเพิ่มไฟล์เข้าไปยัง NHAR file 46
ภาพที่ 3.9 แสดงการทำ reorganization ของ NHAR 47
ภาพที่ 3.10 แสดงตัวอย่างในการเข้าถึงไฟล์ของ NHAR 48
ภาพที่ 3.11 แสดงตัวอย่างในการเพิ่มไฟล์เข้าไปยัง NHAR file 49
ภาพที่ 3.12 (a) แสดงโครงสร้างของไฟล์หลังจากใช้งาน inserting function
(b) แสดงโครงสร้างของไฟล์หลังจากทำการ reorganization 50
ภาพที่ 4.1 แสดงตัวอย่างหน้าจอในระหว่างการสร้าง archive file ของ NHAR 53
ภาพที่ 4.2 แสดงตัวอย่างโครงสร้างข้อมูลที่อยู่ภายใน NHAR directory 54
ภาพที่ 4.3 แสดงตัวอย่างข้อมูลที่อยู่ภายใน index file 54
ภาพที่ 4.4 แสดงตัวอย่างการเพิ่มไฟล์ 54
ภาพที่ 4.5 แสดงตัวอย่างโครงสร้างไฟล์หลังจากเพิ่มไฟล์ 55

ภาพที่ 4.6 กราฟผลการทดลองการวัดประสิทธิภาพโดยแยกตามจำนวน index file 56
ภาพที่ 4.7 กราฟผลการทดลองประสิทธิภาพในการเข้าถึงไฟล์ขนาดเล็ก.. 57
ภาพที่ 4.8 แสดงประสิทธิภาพในการเข้าถึงไฟล์ของ HAR และ NHAR 58
ภาพที่ 4.9 กราฟผลการทดลองการเปรียบเทียบเวลาในการสร้างไฟล์ 59
ภาพที่ 4.10 กราฟแสดงประสิทธิภาพในการเพิ่มไฟล์เข้าไปยังไฟล์ archive ที่มีอยู่แล้ว 60
ภาพที่ 4.11 แสดงประสิทธิภาพในการเพิ่มไฟล์ของ HAR แบบดั้งเดิม.. 61
ภาพที่ 4.12 กราฟแสดงการใช้งานพื้นที่บน NameNode 62
ภาพที่ 4.13 แสดงการเปรียบเทียบการใช้งานพื้นที่บน NameNode ระหว่าง HAR และ NHAR 62

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มาของปัญหา

เนื่องจากการเติบโตอย่างรวดเร็วของอินเทอร์เน็ตทำให้การเติบโตของข้อมูลเพิ่มมากขึ้นไปด้วย ทำให้ Cloud computing ได้รับความสนใจอย่างมากในการจัดเก็บข้อมูล และพัฒนา software และการบริการต่างๆ [1] ซึ่งส่วนสำคัญของ Internet server infrastructure คือ distributed file system ซึ่งระบบที่ได้รับความนิยมในปัจจุบันคือ Google file system (GoogleFS) [2], Hadoop distributed file system (HDFS) [3] และ Amazon Simple Storage Service (S3) [4] โดยที่ HDFS หรือ Hadoop Distributed File System เป็น open source ที่ได้รับอิทธิพลมาจาก GoogleFS ที่ถูกออกแบบมาเพื่อจัดเก็บและทำงานกับข้อมูลที่มีขนาดใหญ่ ซึ่งหากต้องจัดการกับไฟล์ที่มีขนาดเล็ก หรือข้อมูลที่มีขนาดน้อยกว่า HDFS block (ขนาดมาตรฐานคือ 128MB) จำนวนมากจะทำให้ประสิทธิภาพในการทำงานแย่ลง แต่ในปัจจุบันนี้มีข้อมูลขนาดเล็กจำนวนมาก ยกตัวอย่างเช่น ข้อความ Email [5] เนื่องจากทุกบริษัทใช้ email เพื่อติดต่อสื่อสารระหว่างกัน ทำให้ผู้ใช้งานมีข้อความ email จำนวนมากอยู่ใน mailbox ของตัวเอง ซึ่งข้อความเหล่านั้นอาจเป็นข้อความที่สำคัญ และอาจต้องเรียกใช้งานข้อความ email บ่อย ดังนั้นผู้ดูแลระบบจึงจำเป็นที่จะต้องให้ความสำคัญกับการจัดเก็บและการค้นหาข้อความ email เหล่านั้น ซึ่งการนำ Hadoop มาใช้ในการจัดการกับข้อมูล email ถือเป็นทางเลือกที่ดีทางเลือกหนึ่ง เนื่องจาก Hadoop สามารถจัดเก็บข้อมูลได้เป็นจำนวนมาก , มีความทนทานต่อการสูญเสียที่สูง เพราะมีการทำ replication ข้อมูลอัตโนมัติ และยังสามารถดำเนินงานในแบบ parallel ได้เมื่อต้องการอีกด้วย แต่ข้อความ email แต่ละข้อความนั้นมีขนาดเพียง 1 byte ถึง 25 MB เท่านั้น ซึ่งถือว่าเป็นไฟล์ขนาดเล็ก โดยเหตุผลที่ทำให้ HDFS นั้นไม่สามารถจัดการกับไฟล์ขนาดเล็กได้ดีเท่าที่ควรนั้นเนื่องมาจาก HDFS นั้นมีโครงสร้างการทำงานเป็น single master-multiple slave โดยจะมี NameNode ทำหน้าที่เป็น master node เพียงตัวเดียว และการทำงานภายในระบบทั้งหมดจะต้องทำงานผ่าน NameNode ทุกครั้ง ซึ่งในการทำงานปกติ NameNode จะคอยให้บริการ metadata กับผู้ใช้เมื่อมีการร้องขอ โดยจะทำการจัดเก็บ metadata เอาไว้ในหน่วยความจำเพื่อให้มีความรวดเร็วในการทำงาน จึงทำให้เกิดปัญหาตามมาคือ ปัญหาคอขวด หรือ Bottle neck [6] กล่าวคือเมื่อมีการร้องขอการใช้งานจำนวนมาก จะทำให้ NameNode ต้องรับ request ในการจัดเก็บและการกระจาย block บ่อย ส่งผลให้ NameNode ทำงานหนัก และอาจไม่สามารถทำงานได้อย่างมีประสิทธิภาพเท่าที่ควร นอกจากนั้นยังรวมถึงปัญหาทางด้านพื้นที่เก็บข้อมูลบน NameNode มีขีดจำกัดตาม physical memory เนื่องจาก HDFS จัดเก็บ metadata เอาไว้ใน memory ของ NameNode ดังนั้นยิ่งมีไฟล์จัดเก็บในระบบมากเท่าไหร่ ก็ยิ่งต้องใช้พื้นที่ในการจัดเก็บ namespace มากขึ้นเท่านั้น

วิธีที่ใช้ในการจัดการกับไฟล์ขนาดเล็กบน HDFS ในปัจจุบันนั้นสามารถแบ่งออกเป็น 2 วิธี คือ ขยายความสามารถในการรับงานของ NameNode โดยการสร้าง multiple NameNode หรือการทำ Federation เพื่อให้สามารถรองรับงานได้มากขึ้น และการลดการใช้งาน memory ที่

NameNode โดยการนำไฟล์ขนาดเล็กจำนวนมากรวมเข้าด้วยกันให้กลายเป็นไฟล์ขนาดใหญ่และสร้าง index ของแต่ละไฟล์ ในการขยายความสามารถของระบบเพื่อให้รับงานได้มากขึ้นนั้น Apache Hadoop Foundation จึงทำการพัฒนา HDFS Federation [7, 8] ขึ้นมา ซึ่งทำให้ HDFS รองรับการทำงานแบบ multiple namespace ใน cluster เพื่อเพิ่มความสามารถในการขยายระบบ และแยกกันทำงานได้ แต่ปัญหาคือมีความยุ่งยากในการ configure ระบบให้สามารทำงานร่วมกันได้อย่างสอดคล้อง และไม่มีข้อผิดพลาด ในขณะที่ Jiang et al [9], Dong et al [10, 11] นำเสนอการใช้งาน local cache ในการจัดเก็บ metadata บางส่วนเอาไว้ เพื่อลดการใช้งาน NameNode แต่การแก้ไข metadata management เป็นการเพิ่มความสามารถของระบบเพียงบางอย่างเท่านั้น ซึ่งอาจทำให้มีผลกระทบกับส่วนอื่น โดยจากการทดลองแสดงให้เห็นว่าประสิทธิภาพในการเข้าถึงหลังจากทำการแก้ไขแล้วไม่ได้มีประสิทธิภาพที่ดีมากขึ้นเท่าที่ควร ส่วน Liu et al [12] นำเสนอการปรับปรุง IO performance ของข้อมูล Geographic ที่มีขนาดเล็กบน HDFS ซึ่งสามารถใช้ได้กับข้อมูลเฉพาะเท่านั้น ซึ่งในที่นี้คือ Geographic data และไม่สามารถนำมาใช้กับข้อมูลชนิดอื่นได้ นอกจากนั้นยังไม่มีกลไกที่ช่วยในการเพิ่มประสิทธิภาพให้ดีมากขึ้น นอกจากที่ได้กล่าวไปแล้ว วิธีในการแก้ปัญหาไฟล์ขนาดเล็กบน Hadoop ที่ Hadoop Foundation ทำการพัฒนาขึ้นเพื่อแก้ปัญหายังมี Sequence File และ Hadoop Archive โดยที่ Sequence File [5] เป็น Feature ชนิดหนึ่งที่ใช้ในการจัดเก็บไฟล์ใน HDFS ซึ่งทำการจัดเก็บแบบ binary key-value pairs แต่ปัญหาคือเมื่อต้องการค้นหาไฟล์ต้องเริ่มหาจากต้นไฟล์เสมอ จึงทำให้มีผลกระทบกับประสิทธิภาพในการเข้าถึงไฟล์ รวมถึงเวลาที่ใช้ในการแปลงไฟล์ให้อยู่ในรูปแบบของ Sequence File ยังใช้เวลานานอีกด้วย, Hadoop Archive (HAR)[13] เป็นรูปแบบในการจัดเก็บข้อมูลแบบพิเศษบน HDFS ซึ่งถูกนำมาใช้งานเพื่อแก้ปัญหาการจัดเก็บไฟล์ขนาดเล็กบน HDFS โดยการรวมไฟล์จำนวนมากเหล่านั้นเอาไว้ด้วยกันภายใน HDFS blocks เพื่อลดการใช้งาน namespace ของ NameNode ให้น้อยลง ซึ่ง Mackey et al [14] เสนอการนำ HAR มาใช้เพื่อลดขนาดของ metadata storage requirement ของไฟล์ขนาดเล็ก โดยจากผลการทดลองแสดงให้เห็นว่า HAR สามารถลดการใช้งาน memory ได้อย่างมีประสิทธิภาพ แต่การเข้าถึงไฟล์นั้นค่อนข้างช้ากว่าการเข้าถึงไฟล์จาก HDFS โดยตรง เนื่องจากการใช้งาน HAR นั้นเป็นการเพิ่ม overhead ที่ไม่จำเป็น จึงทำให้ประสิทธิภาพในการเข้าถึงช้าลง

ดังนั้นในงานวิจัยนี้ทางผู้วิจัยจึงนำเสนอกลไกในการเพิ่มประสิทธิภาพในการเข้าถึงบน HDFS ให้ดีมากยิ่งขึ้น โดยสามารถจัดการกับพื้นที่ที่ใช้ในการจัดเก็บ metadata บน NameNode ได้อย่างเหมาะสม ซึ่งจะนำหลักการในการทำงานของ HAR มาใช้เป็นพื้นฐานในการวิจัย ถึงแม้ว่า HAR จะมีข้อจำกัดบางประการ แต่ข้อดีของ HAR คือ เราสามารถนำ HAR มาใช้งานได้โดยไม่ต้องทำการแก้ไขหรือเปลี่ยนแปลงโครงสร้างการทำงานของ HDFS ในขณะที่หากนำ HDFS Federation หรือวิธีการอื่นๆที่กล่าวไปแล้วข้างต้นมาใช้ จะต้องทำการแก้ไขโครงสร้างของ HDFS ซึ่งอาจส่งผลต่อการทำงานอื่นๆภายในระบบ โดยเฉพาะอย่างยิ่ง อาจส่งผลต่อการ upgrade ได้ โดยที่งานวิจัยนี้จะนำเสนอ Hadoop Archive ในรูปแบบใหม่ที่เรียกว่า New Hadoop Archive หรือ NHAR ซึ่งเป็นการปรับปรุงกลไกการทำ index ขึ้นมาใหม่เพื่อให้มีประสิทธิภาพในการจัดการ metadata ของ HDFS และประสิทธิภาพในการเข้าถึงให้ดีมากยิ่งขึ้น โดยไม่ต้องทำการแก้ไขหรือเปลี่ยนแปลงโครงสร้างการทำงานของ HDFS นอกเหนือจากนี้ ในงานวิจัยนี้ยังเพิ่มความสามารถในการทำงานของ HAR โดยการ

ปรับปรุงโครงสร้างการทำงานของ HAR ให้สามารถเพิ่มไฟล์ลงไปไฟล์ archive ที่มีอยู่แล้ว ซึ่ง HAR ดั้งเดิมนั้นไม่สามารถทำได้อีกด้วย

1.2 วัตถุประสงค์

โครงงานวิจัยนี้มีวัตถุประสงค์ คือ เพื่อศึกษาและรับมือกับข้อมูลขนาดเล็กที่เกิดขึ้นจำนวนมากในปัจจุบัน และพัฒนากลไกเพื่อช่วยให้ระบบมีประสิทธิภาพในการเข้าถึงมากยิ่งขึ้น

1.3 ขอบเขตของงานวิจัย

เครื่องมือที่ใช้ในงานวิจัยนี้ประกอบด้วย

- CPU: Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz Processors, RAM: 4 GB, OS: 11.10 (GNU/Linux 3.0.0-26-server x86_64) จำนวน 5 เครื่อง โดยแบ่งเป็น NameNode 1 เครื่องและ DataNode 4 เครื่อง
- Hadoop version 0.20.2-cdh3u5
- Har File System 0.20.2-cdh3u5
- Java version 1.6.0_24
- HDFS block size มีขนาด 128 MB และมีกำหนดให้ replica มีจำนวน 3 ไฟล์
- ไฟล์ตัวอย่างที่ใช้ในการทดสอบเป็นข้อมูล log และไฟล์ text ที่มีขนาดตั้งแต่ 2kB ถึง 40MB จำนวน 80,000 ไฟล์ ขนาดไฟล์รวมทั้งหมดประมาณ 5.6GB
- ทำการทดสอบการเข้าถึงไฟล์โดยการอ่านไฟล์ และเปรียบเทียบกับการเข้าถึง HDFS โดยตรง และ HAR แบบดั้งเดิม

1.4 ประโยชน์ที่คาดว่าจะได้รับ

ประโยชน์ที่คาดว่าจะได้รับในงานวิจัยนี้ ได้แก่

1. เข้าใจโครงสร้างในการจัดเก็บไฟล์ขนาดเล็กบน HDFS ด้วย Har file
2. ได้ความรู้เกี่ยวกับหลักการและกลไกในการเขียนสร้าง index file ที่ใช้ในการค้นหาข้อมูลภายใน Har file
3. ได้ต้นแบบของ Framework ในการจัดการกับข้อมูลขนาดเล็กบน HDFS ที่นำ HAR file มาปรับปรุงเพื่อให้มีประสิทธิภาพในการเข้าถึงข้อมูลมากยิ่งขึ้น
4. สามารถนำความรู้จากผลการวิจัยนี้ไปประยุกต์ใช้จริงต่อไปในอนาคต

1.5 แผนการดำเนินการวิจัย

ในขั้นตอนการดำเนินงานแบ่งออกเป็น 5 ขั้นตอน ดังนี้

1.5.1 ศึกษางานวิจัยที่เกี่ยวข้อง

ทำการศึกษางานวิจัยที่เกี่ยวข้อง เช่น ทำการศึกษาเกี่ยวกับการทำงานของ HDFS, ศึกษาลักษณะการอ่านและการทำ archive file ของ HAR โดยละเอียด เป็นต้น

1.5.2 ศึกษากลไกในการทำ index file

ทำการศึกษาวิธีการและกลไกในการสร้าง index file ที่จะนำมาใช้ในการพัฒนาระบบ เช่น ศึกษาโครงสร้างการทำงาน index file ของ HAR แบบดั้งเดิม เพื่อศึกษาปัญหาที่เกิดขึ้น และทดลองสร้าง index file ขึ้นมาใหม่เพื่อใช้สำหรับการค้นหาข้อมูลภายใน HDFS, ทำการศึกษาการเขียนภาษา Java ที่ใช้ในการเขียนโปรแกรมสำหรับการทำไฟล์ archive และการเข้าถึง Har file เป็นต้น

1.5.3 ขั้นตอนออกแบบและทำการสร้าง Framework การทำงานของระบบ

ทำการออกแบบและสร้าง Framework การทำงานของระบบ

1.5.4 ขั้นตอนการทดลองเพื่อทดสอบความสามารถ

นำ Framework การทำงานไปทำการทดลองใช้งานกับไฟล์ที่มีขนาดและจำนวนรวมที่ต่างกัน และทำการทดสอบการเพิ่มไฟล์เข้าไปยัง archive file ที่มีอยู่แล้ว ว่าสามารถทำได้อย่างมีประสิทธิภาพตามที่คาดหวังเอาไว้หรือไม่ รวมถึงทำการปรับปรุงแก้ไขและประเมินผลการวิจัย

1.5.5 ขั้นตอนการสรุปผลการทดลองและจัดทำวิทยานิพนธ์

นำผลการทดลองที่ได้มาสรุปว่า Framework ที่ได้ทำการสร้างขึ้นมานั้นมีความสามารถและประสิทธิภาพเป็นที่น่าพอใจหรือไม่ พร้อมทั้งหาเหตุผลมาอธิบายถึงผลที่เกิดขึ้น

1.6 ผลงานตีพิมพ์

ส่วนหนึ่งของวิทยานิพนธ์นี้ได้นำเสนอในการประชุมวิชาการ ดังนี้

- Chatuporn Vorapongkitipun and Natawut Nupairoj , Improving Performance of Small-File Accessing in Hadoop , The 11th International Joint Conference on Computer Science and Software Engineering (JCSSE 2014) , Pattaya, Thailand, May 14-16, 2014

บทที่ 2
ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

ในงานวิจัยนี้จะมีทฤษฎีและงานวิจัยที่เกี่ยวข้องที่มีความสำคัญ ซึ่งจะประกอบด้วยองค์ความรู้ด้านระบบแฟ้มข้อมูลแบบกระจายของฮาดูป (Hadoop Distributed File System) และการเก็บไฟล์ของฮาดูปที่เรียกว่า Hadoop Archive ในส่วนของงานวิจัยที่เกี่ยวข้องนั้น จะเป็นวิธีในการแก้ปัญหาไฟล์ขนาดเล็กบน HDFS ที่มีอยู่ในปัจจุบัน

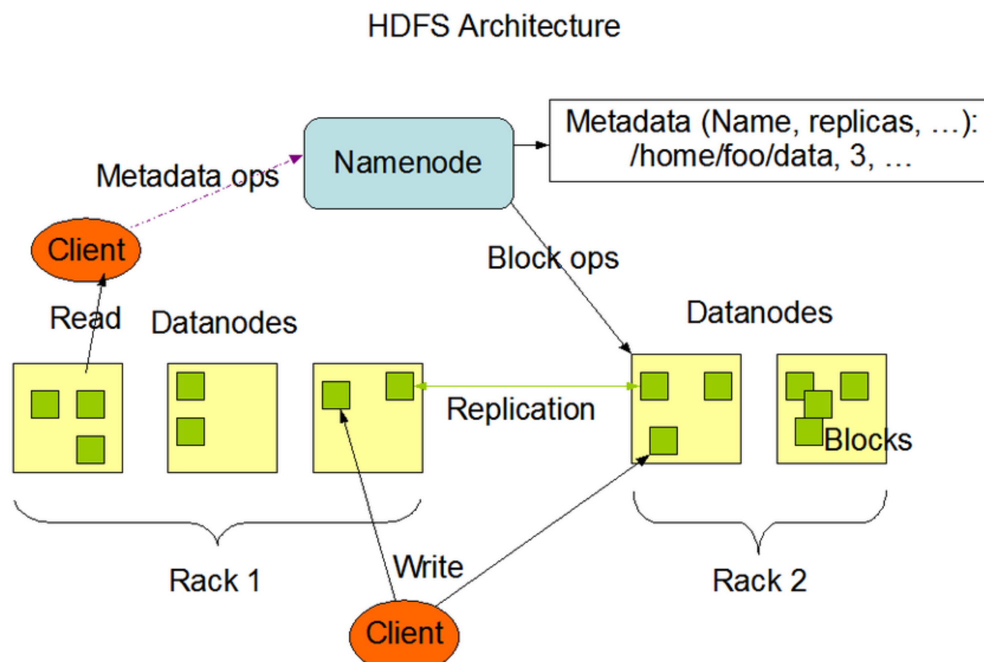
2.1 Hadoop Distributed File System (HDFS) [3, 15]

Hadoop Distributed File System หรือ HDFS นั้นเป็น distributed file system ที่ออกแบบมาเพื่อทำงานบน commodity hardware ซึ่งมีความคล้ายคลึงกับ distributed file system อื่นๆ แต่มีความแตกต่างจากระบบปฏิบัติการอื่นๆตรงที่ HDFS สามารถทนต่อการสูญเสียสูง (highly fault-tolerant) และออกแบบมาเพื่อรองรับการทำงานของ low-cost hardware โดยที่ HDFS จะรองรับการทำงานแบบ batch processing มากกว่าการทำงานแบบ real-time ซึ่งทำให้มี high throughput ในการเข้าถึง application data และมีความเหมาะสมสำหรับ application ที่มีข้อมูลขนาดใหญ่ ซึ่งสามารถมีได้ถึงหนึ่งหมื่นโหนด, หนึ่งร้อยล้านไฟล์ และมีขนาดใหญ่ได้มากถึง 10 petabyte

โดยปกติแล้ว HDFS จะเป็นแบบ write-once-read-many access model โดยไฟล์จะถูกเขียน เพียงครั้งเดียว และจะไม่สามารถแก้ไขไฟล์ได้ ซึ่งทำให้ความสัมพันธ์ต่างๆของข้อมูลมีความง่ายมากขึ้น และทำให้สามารถเข้าถึงข้อมูลได้แบบ high throughput นอกจากนั้นแล้ว HDFS ยังมีการย้ายการประมวลผลให้ไปอยู่ใกล้กับที่เก็บข้อมูลอีกด้วย ซึ่งจะช่วยลดความคับคั่งของเครือข่าย และยังช่วยเพิ่ม throughput ให้กับระบบได้อีกด้วย

2.1.1 สถาปัตยกรรมของ HDFS

HDFS มีโครงสร้างการทำงานเป็นแบบ master/slave โดยจะมีโหนดที่สำคัญอยู่ 2 ชนิด คือ NameNode จำนวน 1 โหนด และ DataNode จำนวนหนึ่ง



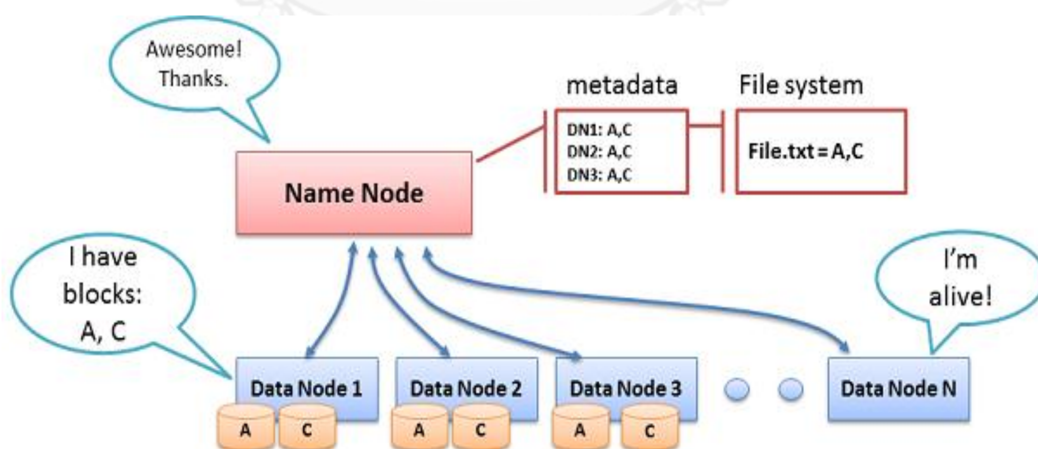
ภาพที่ 2.1 แสดงสถาปัตยกรรมของ HDFS[15]

2.1.1.1 NameNode

เป็น master server ที่คอยจัดการ file system namespace เช่น เปิด-ปิดไฟล์, เปลี่ยนชื่อไฟล์และสารบบ (directories) นอกจากนั้นยังทำหน้าที่ตัดสินใจในการ mapping block ว่าจะเก็บเอาไว้ที่ DataNode ใด ซึ่งไฟล์และ directories จะถูกแสดงอยู่บน NameNode ในรูปของ inode ซึ่งจะเก็บข้อมูลเกี่ยวกับ permissions, การแก้ไขต่างๆ และเวลาในการเข้าถึง, namespace และ disk space quotas โดยที่ไฟล์จะถูกแบ่งออกเป็น blocks (โดยปกติจะมีขนาด 128 megabytes แต่สามารถกำหนดเองในแต่ละไฟล์ได้) และแต่ละ block จะถูกทำ replicate ไปตาม DataNode ต่างๆ (โดยปกติจะทำ replicate 3 โหนด แต่สามารถกำหนดเองได้ในแต่ละไฟล์) โดย NameNode จะจัดเก็บ namespace ทั้งหมดเอาไว้ใน RAM ซึ่งในแต่ละไฟล์จะมี inode data และรายชื่อของ blocks โดยจะจัดเก็บ metadata ทั้งหมดเอาไว้ใน Namespace image ซึ่งจัดเก็บเอาไว้ใน local host ของ file system ที่เรียกว่า checkpoint นอกจากนั้น NameNode ยังมีการเก็บ Journal ซึ่งเป็น log ที่จะเก็บการเปลี่ยนแปลงต่างๆภายในระบบเอาไว้ใน local host file system และเพื่อเพิ่มความทนทาน จะมีการทำ redundant copies ของ checkpoint และ journal ที่ server อื่นๆ เอาไว้ด้วย ซึ่งระหว่างที่ทำการ restart NameNode จะทำการ restore namespace โดยอ่านจาก namespace และ journal

2.1.1.2 DataNode

เป็น one per node ภายใน cluster ซึ่งจะคอยจัดการการจัดเก็บข้อมูลไปยังโหนดที่ทำงานอยู่ ซึ่งจะรับคำสั่งต่างๆมาจาก NameNode และทำตามคำสั่งนั้น นั่นคือทำหน้าที่ในการสร้าง-ลบ block และการทำ replication โดยในแต่ละ block replica บน DataNode จะประกอบด้วยไฟล์สองไฟล์ โดยไฟล์แรกจะเป็นข้อมูล และไฟล์ที่สอง คือ block ของ metadata ซึ่งรวมถึง checksum ของ block data และ block's generation stamp โดยขนาดของ data file จะเท่ากับความกว้างของ block จริงๆและไม่มีการร้องขอพื้นที่พิเศษเพื่อใช้ในการปัดให้ครบตามขนาดเหมือน file system แบบดั้งเดิม โดยในระหว่างการ startup แต่ละ DataNode จะเชื่อมต่อไปยัง NameNode และทำการ handshake เพื่อยืนยัน namespace ID และ software version ของ DataNode หากไม่ตรงกับ NameNode ก็จะทำการปิดการทำงานของ DataNode นั้นอัตโนมัติเพื่อไม่ให้เกิด data corruption หรือ การสูญหายของข้อมูล หลังจากนั้นจะทำการส่ง block report ไปยัง NameNode ซึ่งจะประกอบด้วย block id, generation stamp และ ความยาวของแต่ละ block replica โดย block report จะทำการส่งทุกหนึ่งชั่วโมง นอกจากนั้นแล้ว ในระหว่างการทำงานปกติ DataNode จะส่ง heartbeats ไปยัง NameNode เพื่อยืนยันว่า DataNode นั้นยังทำงานอยู่และ block replicas ภายในก็ยังคงทำงานอยู่เช่นเดียวกัน ซึ่งข้อมูลที่ถูกส่งไปใน heartbeats นั้นจะมีข้อมูลเกี่ยวกับ ความจุ, storage ที่ถูกใช้งาน และจำนวนของ data transfer ที่กำลังทำงานอยู่ ซึ่งข้อมูลเหล่านี้จะใช้สำหรับการจองพื้นที่และการตัดสินใจในการทำ load balancing ของ NameNode โดยปกติจะทำ heartbeat ทุก 3 วินาที หาก NameNode ไม่ได้รับ heartbeat จาก DataNode ภายใน 10 นาที NameNode ก็จะทำการเปลี่ยนสถานะของ DataNode นั้นให้เป็น out-of-service และจะสร้าง replicas ที่อยู่ใน DataNode นั้นไปไว้ใน DataNode ใหม่

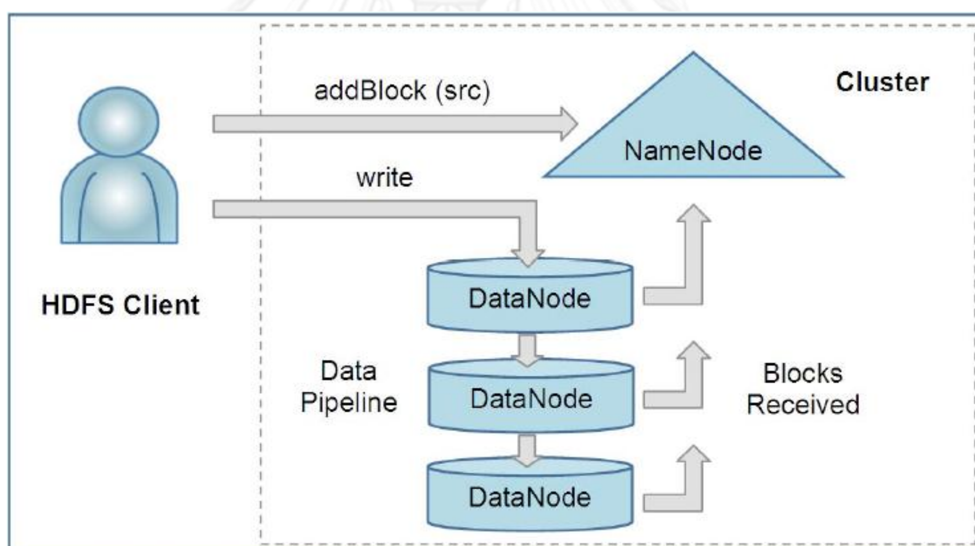


ภาพที่ 2.2 แสดงการส่ง heartbeat ระหว่าง NameNode และ DataNode[16]

นอกจากนั้น HDFS ยังเพิ่มความสามารถอื่นๆเพื่อให้ระบบมีความน่าเชื่อถือเพิ่มมากขึ้น ได้แก่ CheckpointNode และ BackupNode โดย CheckpointNode นั้นจะนำ checkpoint ล่าสุดและ journal ทั้งหมดจาก NameNode รวมเข้าด้วยกัน เพื่อสร้าง checkpoint ใหม่ขึ้นมาและส่งค่าคืนกลับไปให้ NameNode โดยปกติแล้วมักจะไม่รันอยู่บนเครื่องเดียวกับ NameNode แต่ต้องการหน่วยความจำเท่ากับ NameNode ซึ่งการสร้าง checkpoint เป็นครั้งคราวนั้นถือเป็นทางหนึ่งที่ใช้ในการป้องกันการสูญหายของ file system metadata โดยจะทำให้ระบบสามารถเริ่มต้นจาก checkpoint ล่าสุดได้ถึงแม้ว่าข้อมูลอื่นๆของ namespace image หรือ journal ไม่สามารถใช้งานได้ ในส่วนของ BackupNode นั้น จะมีการเชื่อมต่อเพื่อดูสถานะของ NameNode ตลอดเวลา โดยทำหน้าที่จัดเก็บสถานะล่าสุดของ NameNode เอาไว้ ซึ่งภายในจะประกอบด้วยไฟล์ metadata ทั้งหมด ยกเว้นสถานที่เก็บ block

2.1.2 การเขียนไฟล์ของ HDFS

HDFS นั้นรองรับการอ่าน, เขียน และลบไฟล์ รวมถึงการสร้างและลบ directories ด้วย



ภาพที่ 2.3 แสดงการอ่านและเขียนไฟล์ของ HDFS [16]

2.1.2.1 การอ่านไฟล์

HDFS client ต้องทำการติดต่อไปยัง NameNode เพื่อร้องขอที่อยู่ของ data block ที่ต้องการก่อนว่าถูกจัดเก็บเอาไว้ที่ DataNode ใดบ้าง จากนั้นจึงทำการเชื่อมต่อไปยัง DataNode ที่อยู่ใกล้กับ client ที่สุดเพื่อทำการอ่าน block นั้นๆ

2.1.2.2 การเขียนไฟล์

HDFS client ต้องทำการร้องขอไปยัง NameNode เพื่อให้ NameNode ทำการเลือก DataNode ที่จะใช้ในการทำ block replica จำนวนสาม DataNode จากนั้นจึงทำการเขียน data ไปยัง DataNode ที่ถูกเลือกแบบ pipeline โดยจะต้องทำการส่ง heartbeat ไปบอกกับ NameNode เป็นระยะๆ เพื่อให้รู้ว่ายังทำงานอยู่ และเมื่อทำการเขียนเสร็จสิ้นและไฟล์ถูกปิดเรียบร้อยแล้ว client อื่นๆก็จะสามารถเข้าถึงไฟล์นั้นๆได้ต่อไป

2.1.3 Replica Placement

ในการวาง replica นั้นมีผลกับความน่าเชื่อถือและประสิทธิภาพของ HDFS ซึ่งถือเป็นข้อแตกต่างกับ distributed file system อื่นๆ ซึ่งจุดประสงค์ของการจัดวางนั้นเพื่อเพิ่มความน่าเชื่อถือ, สภาพพร้อมใช้งาน และการใช้ network bandwidth อย่างเหมาะสม ซึ่ง HDFS นั้นจะทำงานบน cluster ซึ่งส่วนใหญ่จะมีการกระจายไปบน rack ต่างๆ ดังนั้นการติดต่อสื่อสารระหว่างโหนดใน rack ที่ต่างกันจำเป็นจะต้องผ่าน switch ซึ่งโดยปกติแล้ว network bandwidth ที่อยู่ใน rack เดียวกันจะดีกว่า network bandwidth ที่อยู่ต่าง rack โดยปกติแล้วการวาง replica ของ HDFS นั้นจะพยายามให้ต้นทุนในการเขียนนั้นมีค่าน้อยที่สุด และมีความน่าเชื่อถือของข้อมูล, สภาพพร้อมใช้งาน และ bandwidth รวมในการอ่านมากที่สุด โดยเมื่อ block ใหม่ถูกสร้างขึ้น HDFS จะวาง replica แรกบนโหนดเดียวกับผู้เขียน replica ที่ 2 และ 3 จะกระจายอยู่ตามโหนดอื่นใน rack ที่ต่างกัน โดยแต่ละ replica จะไม่ถูกวางเอาไว้ในโหนดเดียวกัน และในแต่ละ rack จะมี replica ไม่เกิน 2 replica กล่าวคือ หาก replicate เป็น 3 ก็จะวาง 2 replica แรกไว้ใน rack เดียวกันแต่อยู่คนละโหนด ส่วนอีก replica จะถูกวางเอาไว้ในอีก rack หนึ่ง

โดยนโยบายในการวาง replica นี้จะช่วยลด inter-rack และ inter-node สำหรับระยะทางในการเขียน และเพิ่มประสิทธิภาพในการเขียนทั่วๆไป เนื่องจากโอกาสที่จะเกิด rack failure นั้นมีน้อยมากกว่า node failure ซึ่งนโยบายนี้จะไม่มีผลกระทบกับความน่าเชื่อถือของข้อมูล ซึ่งในกรณีปกติที่มีการทำ replicate 3 โหนดนั้น จะช่วยลดการใช้งาน network bandwidth เมื่อมีการอ่านข้อมูลใน block ที่อยู่ใกล้กับผู้ใช้งานมากที่สุด

2.2 Hadoop Archive (HAR) [13]

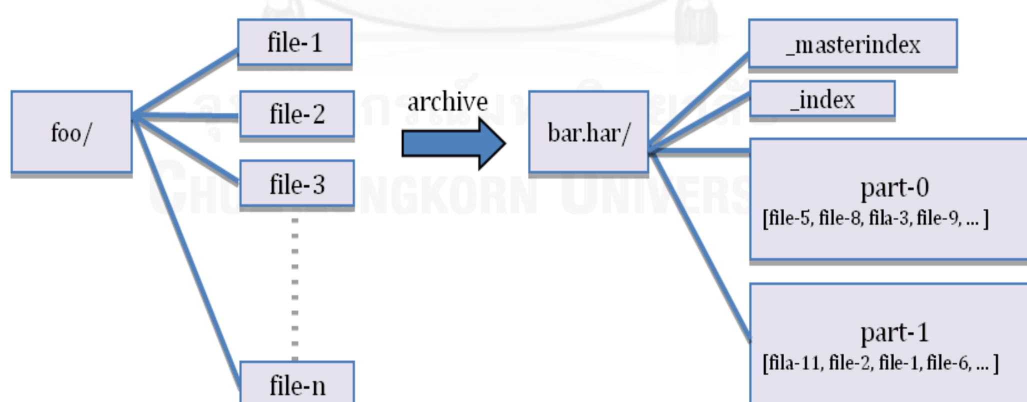
Hadoop Archive หรือ HAR เป็นรูปแบบในการจัดเก็บข้อมูลแบบพิเศษ ซึ่งวัตถุประสงค์หลักในการใช้งาน Hadoop archive คือเพื่อลดการใช้งาน namespace ของ NameNode ให้น้อยลง โดยการรวมไฟล์จำนวนมากเข้าด้วยกันเพื่อให้มีจำนวนน้อยลงและยังสามารถเข้าถึงไฟล์นั้นๆ ได้ง่ายและมีประสิทธิภาพอีกด้วย ซึ่ง Hadoop archive นั้นจะถูกเก็บเอาไว้ใน HDFS directory และลงท้ายด้วย .har เสมอ

การใช้งาน HAR นั้นสามารถแก้ปัญหาเกี่ยวกับไฟล์ขนาดเล็กใน Hadoop ได้อย่างมีประสิทธิภาพ เนื่องจาก HAR นั้นจะรวมไฟล์ขนาดเล็กเข้าไว้ด้วยกันให้กลายเป็นไฟล์ขนาดใหญ่ และสามารถเข้าถึงไฟล์แบบ parallel transparently ได้อย่างมีประสิทธิภาพ ซึ่ง HAR จะช่วยเพิ่มความสามารถในการ scale ระบบโดยการลดการใช้งาน namespace และลด operation load ใน NameNode ซึ่งการปรับปรุงนี้จะเป็นการเพิ่มประสิทธิภาพของ memory ใน NameNode และกระจายการจัดการ namespace ให้เป็นแบบ multiple NameNode

Hadoop Archive นั้นประกอบด้วย 3 ส่วนคือ data model ที่จะกำหนด archive format, FileSystem interface ที่จะใช้ในการเข้าถึงแบบ transparent และ tool ที่ใช้ในการสร้าง archive ด้วย MapReduce jobs

2.2.1 The Data Model: har format

Hadoop Archive’s data format หรือที่เรียกว่า HAR นั้นจะมี โครงสร้างภายในดังภาพที่ 2.4



ภาพที่ 2.4 แสดงการทำ Archive ไฟล์ขนาดเล็กด้วย HAR file [17]

โดยที่;

_masterindex : จัดเก็บ hashes และ offsets

_index : จัดเก็บ file statuses

part-[1..n] : จัดเก็บ file data ที่แท้จริง

โดย file data จะถูกเก็บเอาไว้ใน multiple part files และมีการทำ index เพื่อรักษาต้นฉบับของ data ที่ถูกแยกออกจากกัน ซึ่ง index file นั้นจะมี 2 ส่วนคือ _masterindex และ _index โดยที่ _masterindex จะจัดเก็บ offset ในการเข้าไปยัง _index file เพื่อให้ง่ายต่อการค้นหาเข้าไปยัง _index file และทำให้มีความรวดเร็วในการค้นหา file ส่วน _index file นั้นจะประกอบด้วยชื่อของไฟล์ที่เป็นส่วนหนึ่งของ archive และสถานที่จัดเก็บ file ภายใน part file ซึ่ง part files จะสามารถเข้าถึงได้ด้วย MapReduce program ใน parallel โดย index file นั้นจะมีการเก็บ original directory tree structure และ file status เอาไว้ด้วย โดยในภาพที่ 2.4 แสดง directory ที่ประกอบด้วยไฟล์ขนาดเล็กจำนวนมากที่ถูก archive เข้าไปยัง directory เดียวกันเพื่อให้กลายเป็นไฟล์ขนาดใหญ่และมีการทำ index เพื่อแสดงตำแหน่งของ data ของแต่ละ file

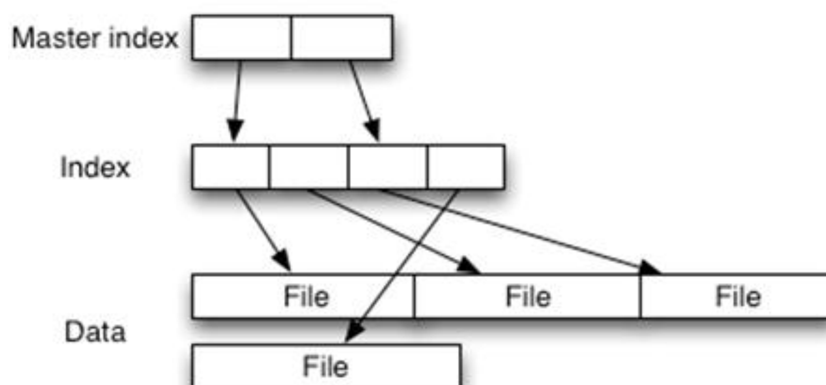
2.2.2 File system Interface: transparent access

ใน archive system ส่วนใหญ่ เช่น tar นั้นจะเป็น tool ที่ใช้ในการทำ archive และ de-archive โดยปกติแล้วจะไม่เหมาะกับการทำงานกับ file system layer จริงและไม่ transparent กับ application writer ซึ่ง user จะต้องทำการ de-archive ไฟล์ก่อนถึงจะสามารถใช้งานได้ ซึ่งต่างจาก Hadoop Archive ที่จะผสมผสานกับ Hadoop’s FileSystem interface โดยที่ HarFileSystem นั้นจะใช้ FileSystem interface ที่สามารถเข้าถึงได้โดยผ่าน har:// scheme ซึ่งจะเป็นการแสดง archive file และ directory tree structure โดยจะ transparent กับ user กล่าวคือ file ใน har จะสามารถถูกเข้าถึงได้โดยไม่ต้อง expand file ยกตัวอย่างเช่น คำสั่งที่ใช้ในการคัดลอกไฟล์จาก HDFS ไปยัง local directory จะใช้คำสั่ง

“hadoop fs –get /foo/file-1 localdir”

โดยสมมติว่า archive bar.har ถูกสร้างขึ้นจาก foo directory ดังนั้นจะสามารถทำการคัดลอก file har ได้โดยใช้คำสั่ง

“hadoop fs –get har:///foo/bar.har/file-1 localdir”

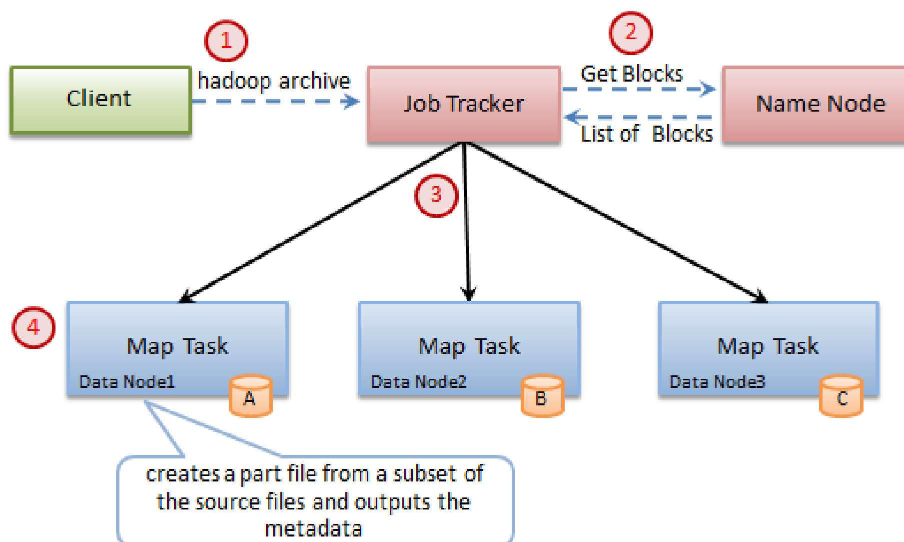


ภาพที่ 2.5 แสดง HAR File Layout [18]

ในการอ่านไฟล์ใน HAR นั้นมีประสิทธิภาพไม่ดีเท่ากับการอ่านไฟล์จาก HDFS โดยตรง เนื่องจากในการเข้าถึง HAR file จะต้องทำการอ่านจาก index file 2 ครั้งดังภาพที่ 2.5 โดยจะเริ่มจากการเข้าค้นหาไฟล์ภายใน Masterindex ก่อนเพื่อดูว่าควรจะเริ่มต้นอ่านไฟล์ภายใจ Index จากที่ใด จากนั้นจึงเริ่มเข้าค้นหาไฟล์จาก Index อีกครั้งโดยเริ่มค้นหาจากตำแหน่งที่ได้รับมาจาก Masterindex และไล่หาไปทีละบรรทัดภายใน Index เมื่อเจอไฟล์ที่ต้องการแล้วก็จะสามารถดึงไฟล์เนื้อหาที่ต้องการจาก Part file ได้ ซึ่งประสิทธิภาพในการเข้าถึงของ Masterindex สำหรับกรณีที่เป็น worst case นั้น จะมีประสิทธิภาพเป็น O(n/1,000) ในขณะที่ Index จะมีประสิทธิภาพคือ O(1,000) ยกตัวอย่างเช่น หากมีไฟล์ในระบบทั้งหมด 10,000 ไฟล์ Masterindex จะต้องเข้าถึงไฟล์ทั้งหมด O(10,000/1,000) นั่นคือ 10 records จากนั้นก็จะต้องไปหาต่อที่ index ไฟล์คือต้องไล่หาอีก 1,000 records ดังนั้นหากไฟล์มีทั้งหมด 10,000 ไฟล์ HAR จะต้องทำการค้นหาทั้งหมด 1,010 records เป็นต้น อย่างไรก็ตามการใช้งาน HAR นั้นถือเป็นการใช้งานที่ดีที่สุดในการทำ archive file ภายใน Hadoop

2.2.3 The Archiving Tool: MapReduce program สำหรับการสร้าง HAR

เครื่องมือที่ใช้ในการทำ archive นั้นจะเป็นการเรียกการใช้งาน MapReduce program ซึ่งมีการทำงาน 2 ส่วนคือ Map และ Reduce โดยที่ Map จะแบ่งไฟล์ออกเป็น map task input โดยในแต่ละ map task จะสร้าง part file จาก subset ของ source file และ output metadata ส่วน Reduce จะรวบรวม part file ทั้งหมด metadata และทำการสร้าง index และ masterindex โดยจะมีขั้นตอนดังต่อไปนี้

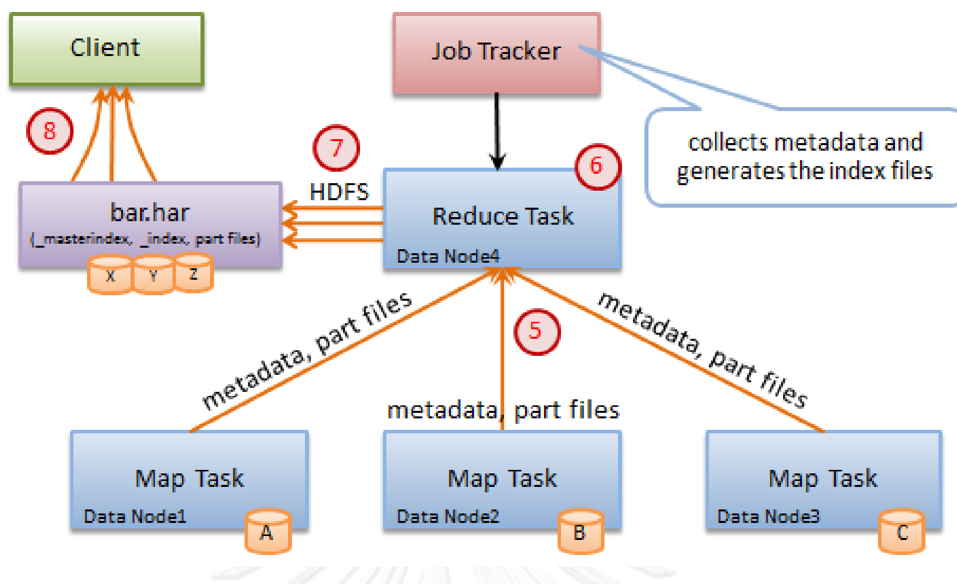


ภาพที่ 2.6 Hadoop Archive - Map Task (adapt from [16])

จากภาพที่ 2.6 จะเป็นขั้นตอนในการทำ Map task เพื่อสร้าง HAR file โดยมีขั้นตอนดังนี้

1. ผู้ใช้ร้องขอไปยัง Job Tracker เพื่อบอกไฟล์ /directory ที่ต้องการทำ HAR file
2. Job tracker ทำการดึงข้อมูลจาก NameNode เพื่อดูสถานที่เก็บไฟล์นั้นๆ
3. ทำการแบ่ง list ของไฟล์ออกเป็น map task input กระจายออกไปบน local data
4. map task จะทำการดึง parent และ list children ภายในออกมาทั้งหมด แล้วทำ HashSet จะทำให้ได้ค่า key และ value แล้วทำการสร้าง part file (ประมาณ 2 GB, แล้วแต่กำหนด) จาก key และ value นั้น (subset ของ source file และ output metadata) เมื่อ Map task ทำงานเสร็จแล้ว แต่ละ node ก็จะทำการจัดเก็บผลลัพธ์ที่ได้เอาไว้ใน temporary local storage ที่เรียกว่า “intermediate data”

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University



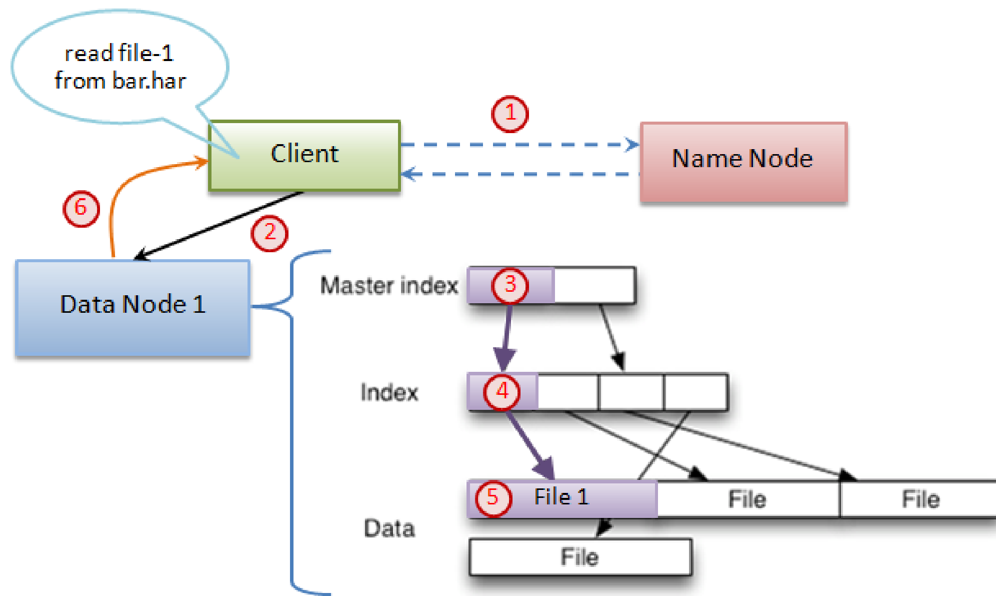
ภาพที่ 2.7 Hadoop Archive - Reduce Task (adapt from [16])

จากภาพที่ 2.7 จะเป็นขั้นตอนในการทำ Reduce task ซึ่งจะทำต่อจากกระบวนการของ Map Task โดยจะทำบน node ใด node หนึ่งภายใน cluster

5. ดึง intermediate data จาก Map task ที่เสร็จสิ้นแล้วทั้งหมด โดย Map task จะส่งผลลัพธ์ไปยังโหนดที่เป็น Reducer 1 ครั้งเท่านั้น
6. ทำการสร้าง index จาก key ของไฟล์ และทุก 1000 index จะทำการสร้าง masterindex ขึ้นมา
7. เขียนผลลัพธ์ที่ได้ทั้งหมดเก็บไว้ในไฟล์ .har บน HDFS โดยจะทำการแยกไฟล์ออกเป็น block ทำ pipeline replication ของแต่ละ block, etc
8. ผู้ใช้สามารถอ่านไฟล์ .har จาก HDFS ได้เหมือนเป็นไฟล์ทั่วไปที่อยู่บน HDFS

2.2.4 การอ่านไฟล์จาก HAR File

ในการเข้าถึงไฟล์ HAR นั้นสามารถเข้าถึงได้โดยไม่ต้องทำการ expand ไฟล์ ซึ่งสามารถเข้าถึงโดยใช้ har:// scheme



ภาพที่ 2.8 Read file from HAR

จากภาพที่ 2.8 เป็นการแสดงการอ่านไฟล์ HAR โดยมีขั้นตอนดังนี้

1. ผู้ใช้ทำการร้องขอไปยัง NameNode เพื่อดูสถานที่เก็บไฟล์ HAR
2. ผู้ใช้ทำการติดต่อไปยัง DataNode ที่มี HAR file อยู่
3. ในการอ่านไฟล์จะเริ่มอ่านจาก Masterindex ก่อนเพื่อดูว่าไฟล์ที่ต้องการนั้นควรจะต้องเริ่มอ่านจาก Index ใด
4. ทำการค้นหาไฟล์โดยเริ่มจาก Index ที่ได้รับมาจาก masterindex เพื่อดูว่าไฟล์อยู่ที่ไหนใน block
5. ทำการอ่านไฟล์ที่ต้องการ
6. ส่งคืนผลลัพธ์กลับไปยังผู้ใช้

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

2.3 MapReduce [19]

MapReduce เป็น framework ในการเขียนโปรแกรมแบบหนึ่งที่ช่วยในงานประมวลผล data set จำนวนมาก หลักการทำงานจะเป็นการทำงานแบบ parallel ซึ่งจะอาศัยเครื่องคอมพิวเตอร์หลายเครื่องช่วยกันทำงาน ซึ่งการทำงานของ MapReduce จะแบ่งเป็น 2 ส่วนหลักคือ Map และ Reduce

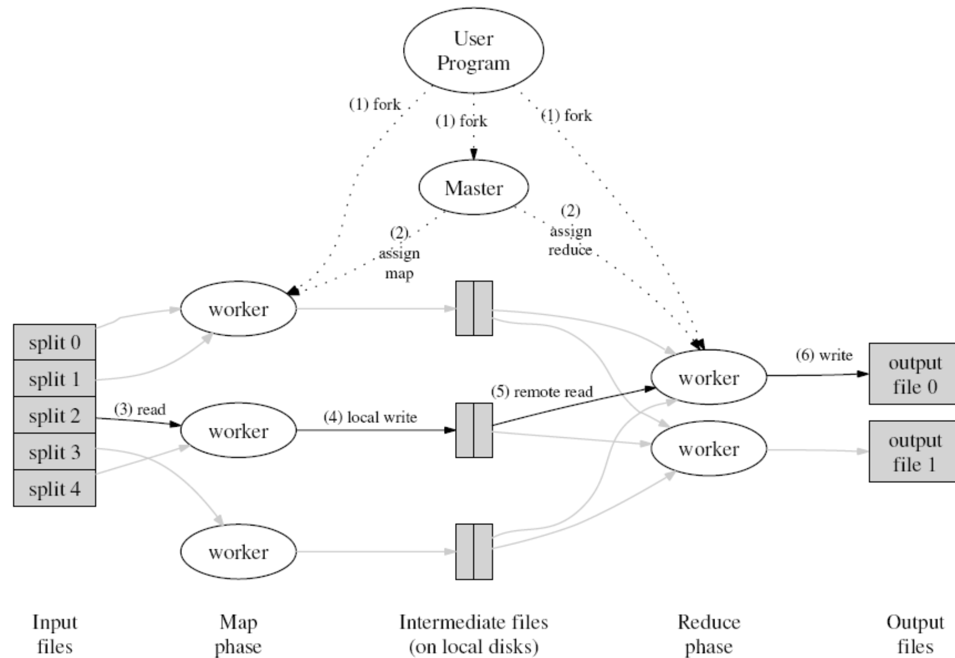
ในขั้นตอนของ Map function นั้น user จะทำการระบุ map function โดย Master node จะนำ input มาแบ่งให้เป็นกลุ่มเล็กๆ และทำการกระจายออกไปตาม worker node โดย worker node จะทำการแบ่ง input ให้เป็น key/value pair เพื่อ generate กลุ่มของ intermediate key/value pair ขึ้นมา ซึ่ง MapReduce library จะทำการรวม intermediate value ที่เกี่ยวข้องกันเอาไว้ด้วยกันและส่งต่อไปยัง Reduce function

ส่วนขั้นตอนของ Reduce function จะรับ intermediate key และกลุ่มของ value สำหรับ key นั้นเอาไว้ แล้วทำการรวม value เหล่านั้นเอาไว้ด้วยกันเพื่อสร้างกลุ่มของ value ที่มีขนาดเล็กลง โดยปกติแล้วจะมีเพียง 0 หรือ 1 output value ที่จะถูกผลิตขึ้นต่อการทำ reduce 1 ครั้ง โดยสามารถรับมือกับ list ของ value ที่มีขนาดใหญ่เกินกว่า memory ได้

2.3.1 Implementation

โดยการทำงานปกติ เมื่อมีการร้องขอการทำงาน MapReduce จะเริ่มต้นจากการ Map โดย input จะถูกกระจายออกไปบน multiple machine และทำการแบ่ง input data แบบอัตโนมัติเพื่อให้อยู่ในกลุ่มของ M splits โดย input splits จะถูกทำงานใน parallel ด้วย machine ที่ต่างกันได้ จากนั้นจะเข้าสู่กระบวนการในการทำ Reduce โดยจะทำการแบ่ง intermediate key space ให้เป็น R pieces โดยใช้ partitioning function (เช่น hash(key) mod R) ซึ่งถูกกำหนดโดย user

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University



ภาพที่ 2.9 แสดงกระบวนการทำงานของ MapReduce [19]

จากภาพที่ 2.9 แสดงกระบวนการทำงานของ MapReduce เมื่อมีการเรียกใช้งานโดย user program โดยมีขั้นตอนในการทำงานดังนี้

1. MapReduce library ใน user program จะเริ่มทำการแบ่ง input files ให้เป็น M pieces จาก 16 MB ให้เป็น 64MB (กำหนดโดย user) จากนั้นจะเริ่มทำการคัดลอก program ไปตามเครื่องต่างๆใน cluster
2. จะมีเครื่องหนึ่งทำหน้าที่เป็น master ส่วนเครื่องอื่นจะทำหน้าที่เป็น worker โดย master จะเลือก idle worker และกำหนดว่าจะให้ทำ map หรือ reduce task
3. worker ที่ถูกกำหนดให้ทำ map task จะอ่าน content ของ input split ที่เหมือนกัน และทำให้เป็น key/value pair จาก input data แล้วส่งต่อไปยัง user-defined Map function โดย intermediate key/value pairs จะถูกผลิตขึ้นโดย Map function ซึ่งจะถูก buffer เอาไว้ใน memory
4. buffer pair จะถูกเขียนเข้าไปยัง local disk และถูกแบ่งให้เป็น R region โดย partitioning จากนั้นจะถูกส่งกลับไปยัง master ที่รับผิดชอบในการส่งต่อไปยัง reduce worker
5. เมื่อ reduce worker ได้รับการแจ้งเตือนจาก master เกี่ยวกับสถานที่เก็บ ก็จะใช้ remote procedure call เพื่ออ่าน buffer data จาก local disk ของ map worker และเมื่ออ่าน intermediate data ทั้งหมดแล้วจะทำการ sort ด้วย intermediate key ดังนั้น key เดียวกันก็จะถูกรวมเอาไว้ด้วยกัน โดยการ sort มีความจำเป็นเพราะปกติ

แล้วจะมี key ที่ต่างกันจำนวนมากที่ map ไปยัง reduce take เดียวกัน ถ้าจำนวนรวมของ intermediate data นั้นใหญ่เกินกว่าที่จะอยู่ใน memory ได้ external sort ก็จะถูกใช้

6. reduce worker จะทำการ sort intermediate data และunique intermediate key แต่ละตัวที่พบซ้ำๆ แล้วทำการส่งต่อ key และกลุ่มของ intermediate value ที่เกี่ยวข้องไปยัง user's Reduce function ซึ่ง output ของ Reduce function จะถูก append เอาไว้เพื่อให้ได้ final output file สำหรับ reduce partition นี้
7. เมื่อ map task และ reduce task ทั้งหมดเสร็จสิ้น master จะดึง user program มาใช้ ซึ่งในตอนนี้ MapReduce จะ call ไปเพื่อใช้งาน user code

2.3.2 Example

ตัวอย่างโปรแกรมที่ใช้ในการอธิบายการทำงานของ MapReduce คือโปรแกรม Word count โดยจะทำการนับจำนวนของคำแต่ละคำที่อยู่ภายใน document ซึ่งจะมีลักษณะการเขียนดังนี้

```
map(String key, String value):
        // key: document name
        // value: document contents
for each word w in value:
                EmitIntermediate(w, "1");
reduce(String key, Iterator values):
        // key: a word
        // values: a list of counts
        int result = 0;
        for each  v in values:
            result += ParseInt(v);
                Emit(AsString(result));
```

จาก code ด้านบน จะเห็นว่า Map function จะปล่อยคำแต่ละคำออกมา บวกด้วยการนับที่เกี่ยวข้องกับเหตุการณ์ที่เกิดขึ้น ('1') ส่วน Reduce function จะรวมการนับทั้งหมดออกมา

2.3.3 การประยุกต์การใช้งาน MapReduce

นอกจากโปรแกรม word count แล้วยังมีการนำ MapReduce ไปใช้เพื่อกับโปรแกรมอื่นอีกจำนวนมาก เพื่อให้การคำนวณต่างๆนั้นง่ายมากขึ้น ยกตัวอย่างเช่น

Distributed Grep : map function จะแสดง line ที่ตรงกับ pattern ส่วน Reduce function จะทำการคัดลอก intermediate data ไปยัง output

Count of URL Access Frequency : map function จะดึง log ของการร้องขอ web page และ output <URL, 1> ส่วน reduce function จะเพิ่ม value ทั้งหมดที่มี URL เดียวกันเอาไว้ด้วยกัน และแสดง <URL, total count> pair

Reverse Web-Link Grape : map function output <target, source> pair ของแต่ละ link ไปยัง target URL ที่พบใน page name source ส่วน reduce function จะนำ list ของ source URL ทั้งหมดที่มี target URL ที่ต้องการเอามาเชื่อมต่อท้ายไปเรื่อยๆ และแสดงออกมาเป็นคู่ <target, list(source)>

Term-Vector per Host : รวม vector ที่มีคำสำคัญส่วนใหญ่ที่ได้รับมาจาก document หรือกลุ่มของ document เหมือนกับ list ของ <word, frequency> pair โดย map function จะแสดง <hostname, term vector> pair ของแต่ละ input document ออกมา (hostname จะถูกดึงออกมาจาก URL ของ document) ส่วน reduce function จะส่งต่อ per-document term vector ที่มี host ที่ได้รับทั้งหมด และเพิ่ม term vector เข้าด้วยกัน และทิ้ง term ที่ไม่ได้ใช้บ่อยออก จากนั้นจะแสดงผลลัพธ์สุดท้าย <hostname, term vector> pair

Inverted Index : map function จะทำการวิเคราะห์แต่ละ document และแสดงลำดับของ <word, document ID> pair ออกมา ส่วน reduce function จะรับ pair ของคำที่ได้รับทั้งหมดแล้วทำการจัดประเภท document ID ที่สัมพันธ์กันและแสดงออกมาเป็น <word, list(document ID)> pair กลุ่มของ output ทั้งหมดจะถูกสร้างขึ้นเป็น simple inverted index ซึ่งมีความง่ายในการคำนวณเพื่อติดตามตำแหน่งที่อยู่ของคำ

Distributed sort : map function จะดึง key จากแต่ละ record และแสดง <key, record> pair ออกมา ส่วน reduce function จะแสดง pair unchanged ทั้งหมด

2.4 Big O (สัญกรณ์โอใหญ่)[20]

ในวิชาทฤษฎีความซับซ้อนและคณิตศาสตร์ สัญกรณ์โอใหญ่ (Big O notation) เป็นสัญกรณ์คณิตศาสตร์ที่ใช้บรรยายพฤติกรรมเชิงเส้นกำกับของฟังก์ชัน โดยระบุเป็นขนาด (magnitude) ของฟังก์ชันในพจน์ของฟังก์ชันอื่นที่โดยทั่วไปซับซ้อนน้อยกว่า สัญกรณ์โอใหญ่เป็นหนึ่งในสัญกรณ์เชิงเส้นกำกับ หรืออาจเรียกว่า สัญกรณ์ของลันเดา หรือ สัญกรณ์ของบัคแมนน์-ลันเดา (ตั้งชื่อตามเอ็ดมุนด์ ลานเดาและเพาล์ บาคมันน์) สัญกรณ์โอใหญ่ใช้ในการเขียนเพื่อประมาณพจน์ในคณิตศาสตร์ ประยุกต์ใช้ในวิทยาการคอมพิวเตอร์เพื่อใช้อธิบายความเร็วประมาณในการทำงานของโปรแกรมในกรณีต้องประมวลผลข้อมูลจำนวนมาก และใช้เพื่ออธิบายประสิทธิภาพของขั้นตอนวิธีหรือโครงสร้างข้อมูลนั้น ๆ

Big O นั้นระบุลักษณะของฟังก์ชันตามอัตราการเติบโต ถึงแม้ฟังก์ชันจะต่างกัน แต่ถ้ามีอัตราการเติบโตเท่ากันก็จะมีสัญกรณ์โอใหญ่เท่ากัน สำหรับสัญกรณ์โอใหญ่แล้ว จะพิจารณาเฉพาะขอบเขตบนของอัตราการเติบโตของฟังก์ชัน อาทิฟังก์ชัน $n^2 + n$ และ $n + 4$ ล้วนมีอัตราการเติบโต *น้อยกว่า*หรือ*เท่ากับ* n^2 นั่นคืออัตราการเติบโตของฟังก์ชัน n^2 เป็นขอบเขตบนของ $n^2 + n$ และ $n + 4$ จึงอาจกล่าวได้ว่า $n^2 + n$ และ $n + 4$ เป็นสมาชิกของเซตของฟังก์ชัน $O(n^2)$ ในขณะที่สัญกรณ์เชิงเส้นกำกับอื่น พิจารณาขอบเขตอื่น ๆ เช่นสัญกรณ์โอเมกาใหญ่พิจารณาขอบเขตล่างของอัตราการเติบโตของฟังก์ชันแทน

โดยที่ Big O จะเป็น upper bound ของ worst case ของ algorithm ซึ่งหมายความว่า algorithm นี้จะทำงานไม่แย่ไปกว่า Big O ของมันแล้ว ซึ่งก็เหมือนกับเป็นตัวบอกถึง ประสิทธิภาพของ algorithm นั้นๆเพื่อใช้ในการเปรียบเทียบ algorithm หลายอันเข้าด้วยกัน โดย algorithm ที่อยู่ใน O(n) หมายความว่าถ้าใช้ algorithm นี้ประมวลข้อมูล n อย่าง จะมีขั้นตอนในการประมวลผลประมาณ n ครั้ง ส่วน algorithm ที่อยู่ใน $O(n^3)$ หมายความว่าถ้าใช้ algorithm นี้ประมวลข้อมูล n อย่าง มันจะมีขั้นตอนในการประมวลผลประมาณ n^3 ครั้ง และไม่ช้าไปกว่านั้น

ด้วยวิธีนี้เราถ้าเรารู้ Big O ของ algorithm ใดๆ เราจะสามารถเปรียบเทียบความสามารถในการแก้ปัญหาของ algorithm นั้นๆได้ อย่างเช่น O(n) นั้นเร็วกว่า $O(n^3)$ ถ้าประมวลข้อมูลเท่าๆกัน โดยเราสามารถเรียงลำดับประสิทธิภาพสูงสุดไปต่ำสุดของ Big O มาตรฐานได้ดังนี้[21]

O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(n^3)$, $O(2^n)$, O(n!)

กล่าวคือ algorithm ที่อยู่ใน O(1) หมายความว่าถ้าใช้ algorithm นี้ประมวลผลข้อมูลจะมีประสิทธิภาพที่ดีมากที่สุด ในขณะที่หากอยู่ใน O(n!) หมายความว่า algorithm นี้จะมีประสิทธิภาพที่แย่ที่สุด

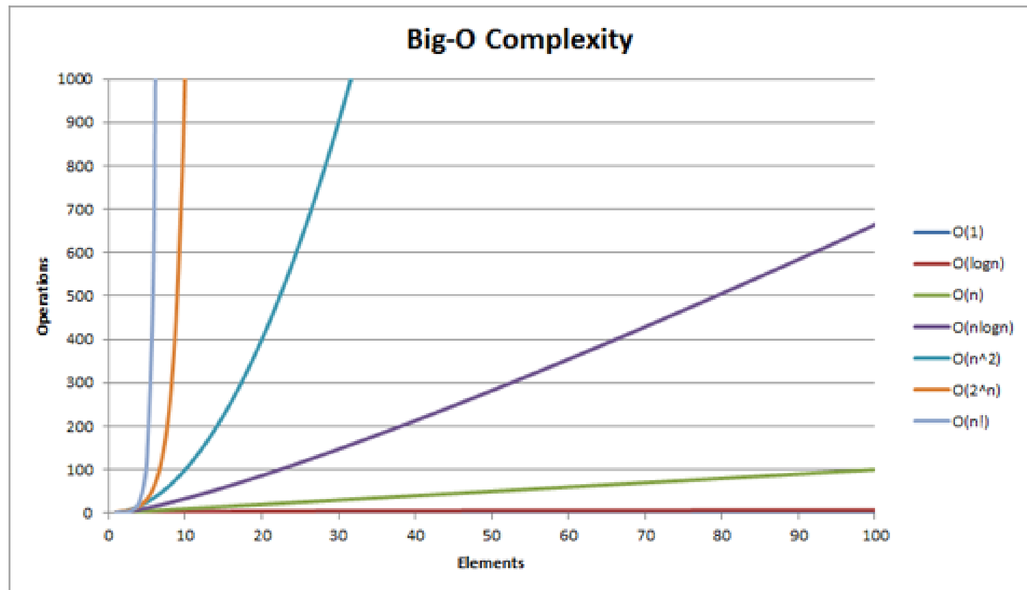
ตารางที่ 1 แสดงตัวอย่างของเวลาที่ใช้ในการ run ของ algorithm ในแต่ละคลาส

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

ตารางที่ 1 แสดงการเปรียบเทียบการรันโปรแกรมเพื่อแก้ปัญหาเดียวกัน โดยมี algorithm ที่แตกต่างกัน โดยเมื่อ n=64 (input 64 ตัว) ถ้า algorithm อยู่ใน class O(log n) งานชิ้นนี้ก็จะถูกทำเสร็จใน 6 นาโนวินาที แต่ในขณะเดียวกัน ถ้าแก้ปัญหาเดียวกันนี้ จำนวน input เท่ากันนี้ แต่เราใช้ algorithm ที่อยู่ใน $O(2^n)$ งานชิ้นนี้จะทำเสร็จในเวลาถึง 5.85 ศตวรรษ เรียกได้ว่า ถ้าเราวิเคราะห์แล้วว่า algorithm ของเราอยู่ใน class นี้ จำนวน input ก็จะต้องมีน้อยกว่า 64 ตัว

โดย order of growth คือ ถ้าเราเพิ่ม input ขึ้นเรื่อยๆจะเห็นได้ว่า เวลาที่ใช้ในการประมวลผลของ Class O(log n) เพิ่มขึ้นน้อยมาก แต่ ในขณะที่ algorithm class อื่นๆนั้น เพิ่มเวลาขึ้นอย่างมาก โดยเฉพาะ class O(n3) นั้นเพิ่มมากที่สุด ดังแสดงอยู่ในภาพที่ 2.10

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University



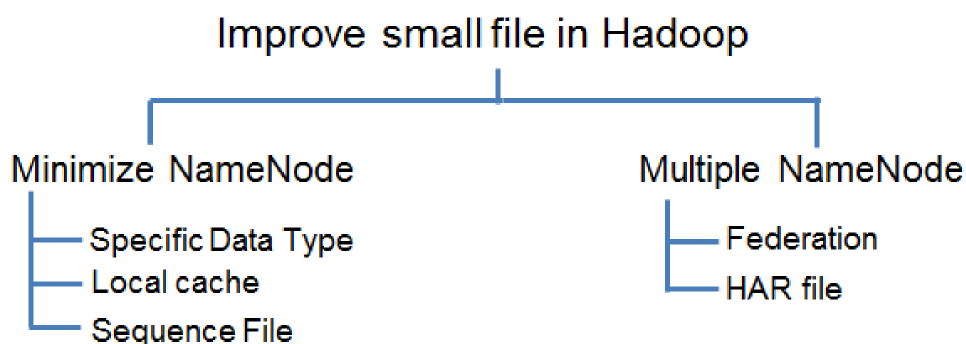
ภาพที่ 2.10 แสดง order of growth แต่ละ class ของ Big O

2.5 ไฟล์ขนาดเล็กบน Hadoop

ในงานวิจัยนี้จะกำหนดให้ขนาด block ของ HDFS มีค่าเป็น 128 MB ดังนั้นหากไฟล์ที่มีขนาดเล็กกว่า 128 MB ในงานวิจัยนี้จะถือว่าเป็นไฟล์ขนาดเล็ก โดยที่หากจัดเก็บไฟล์ขนาดเล็กจำนวนมากเอาไว้ใน HDFS จะทำให้การทำงานภายใน HDFS นั้นไม่มีประสิทธิภาพเท่าที่ควร โดยไฟล์และ block จะถือว่าเป็น name object ใน HDFS ซึ่งจะต้องใช้งานพื้นที่บน NameNode เพื่อจัดเก็บ และพื้นที่จัดเก็บบน NameNode นั้นจะถูกจำกัดตาม physical memory โดยที่ข้อมูลในปัจจุบันนั้นส่วนใหญ่จะเป็นไฟล์ขนาดเล็ก ยกตัวอย่างเช่นหนึ่งใน production cluster ของ Yahoo! [17] นั้นมีไฟล์ที่มีขนาดเล็กกว่า 128 MB มากถึง 57 ล้านไฟล์ ซึ่งเมื่อทำการจัดเก็บเอาไว้ใน HDFS จะทำให้ต้องใช้พื้นที่จัดเก็บบน NameNode มากถึง 95% แต่ใช้พื้นที่บน disk เพียงแค่ 30% ซึ่งหากทำการเปรียบเทียบกับไฟล์ขนาดใหญ่ในกรณีที่ขนาดโดยรวมเท่ากันนั้น พื้นที่จัดเก็บบน NameNode อาจใช้งานเพียง 30% เท่านั้น จะเห็นว่ายิ่งจำนวนไฟล์ขนาดเล็กมีมากเท่าใด ก็ยิ่งต้องใช้พื้นที่ในการจัดเก็บ metadata บน NameNode มากขึ้นเท่านั้น และเนื่องจากพื้นที่บน NameNode นั้นมีขีดจำกัด ดังนั้นหากไฟล์ขนาดเล็กมีจำนวนมากเกินไปก็อาจทำให้พื้นที่บน NameNode เต็มได้

2.6 งานวิจัยที่เกี่ยวข้อง

แนวคิดทั่วไปในการแก้ปัญหาไฟล์ขนาดเล็กบน Hadoop คือการนำไฟล์ขนาดเล็กจำนวนมากมารวมเข้าด้วยกันให้กลายเป็นไฟล์ขนาดใหญ่และสร้าง index ของแต่ละไฟล์ โดยในปัจจุบันนั้นสามารถแบ่งออกเป็น 2 วิธี คือลดการใช้งาน NameNode หรือขยายความสามารถในการรับ load ของ NameNode ให้สามารถรับ load ได้มากขึ้น



ภาพที่ 2.11 แสดงวิธีในการแก้ปัญหาไฟล์ขนาดเล็กบน Hadoop

2.6.1 Minimize NameNode

จากภาพที่ 2.10 แสดงให้เห็นถึงวิธีในการเพิ่มประสิทธิภาพในการทำงานของ Hadoop สำหรับไฟล์ขนาดเล็ก โดยในการลดการใช้งาน NameNode จะมีแนวคิดพื้นฐานคือการรวมไฟล์ที่มีขนาดเล็กให้กลายเป็นไฟล์ขนาดใหญ่ สามารถแบ่งวิธีการทำได้ดังนี้

2.6.1.1 Local cache

ในส่วนนี้จะเป็นการนำ local cache มาใช้เพื่อจัดเก็บ metadata บางส่วนเอาไว้ เพื่อลดการใช้งาน NameNode เมื่อมีการเข้าถึงไฟล์ Jiang et al [9] นำเสนอการปรับปรุง HDFS I/O feature ให้มีความเหมาะสมในการใช้งานไฟล์ขนาดเล็ก โดยจะดูว่า block ไหนมีพื้นที่ว่าง ก็จะทำการเขียนต่อลงไปใน block นั้น เพื่อให้สามารถเก็บไฟล์หลายไฟล์ใน block 1 block ได้ และมีการเก็บ meta-data บางอย่างของไฟล์ (DataNode location) เอาไว้ที่ memory ของ client เพื่อให้ client สามารถเรียกใช้งานจาก DataNode ที่ใช้งานบ่อยได้โดยไม่ต้องทำการติดต่อไปยัง NameNode ส่วน Dong et al [10] นำเสนอการรวมไฟล์ PPT courseware บน internet ซึ่งมีขนาดเล็กมารวมเอาไว้ด้วยกันให้เป็นไฟล์ขนาดใหญ่ใหญ่เพื่อลดภาระ metadata บน NameNode และเพื่อเพิ่มประสิทธิภาพในการจัดเก็บไฟล์ขนาดเล็ก แล้วทำการสร้าง local index file ของแต่ละไฟล์เอาไว้ด้วย โดยใช้ two-level prefetching ที่ซึ่งประกอบด้วย local index file prefetching และ correlated file prefetching เพื่อเพิ่มประสิทธิภาพในการเข้าถึง file ต่อมาในปี 2007 Dong et al [11] นำเสนองานวิจัยที่ทำการพัฒนาต่อจากงานวิจัยเดิม โดยนำ file merging และ file

grouping มาใช้เพื่อพิจารณาว่าหากไฟล์มีความเกี่ยวข้องกันจะทำการจัดเก็บเอาไว้ใกล้กัน นอกจากนั้นยังนำ three-level prefetching และ caching มาใช้เพื่อให้การทำงานกับไฟล์ขนาดเล็กนั้นมีประสิทธิภาพมากยิ่งขึ้น แต่การแก้ปัญหาด้วย local cache นั้นต้องทำการแก้ไข metadata management ซึ่งเป็นการเพิ่มความสามารถของระบบเพียงบางอย่างเท่านั้น (ลดการใช้งาน NameNode และเพิ่มประสิทธิภาพในการเข้าถึง) ซึ่งอาจทำให้มีผลกระทบกับส่วนอื่นที่ต้องใช้งาน metadata management นี้ด้วย โดยจากการทดลองแสดงให้เห็นว่าประสิทธิภาพในการเข้าถึงหลังจากทำการแก้ไขแล้วไม่ได้ดีขึ้นมากนัก

2.6.1.2 Specific Data Type

Liu et al [12] นำเสนอการปรับปรุง IO performance ของข้อมูล Geographic ที่มีขนาดเล็กบน HDFS ให้สามารถใช้งานได้อย่างมีประสิทธิภาพ โดยทำการวมไฟล์ขนาดเล็กให้กลายเป็นไฟล์ขนาดใหญ่เพื่อลดจำนวนไฟล์ และสร้าง index สำหรับแต่ละไฟล์ โดยที่ไฟล์จะถูกรวมและจัดเก็บต่อเนื่องตามลำดับใน physical memory ของ geographic location นอกจากนั้นยังทำการจัดเก็บไฟล์ข้างเคียงและ history spatial data ของ WebGIS เป็น version เอาไว้ด้วย แต่การแก้ปัญหาในงานวิจัยนี้สามารถใช้ได้กับข้อมูลเฉพาะ (geographic data) เท่านั้น ไม่สามารถนำมาใช้กับข้อมูลชนิดอื่นได้ และไม่มีกลไกที่ช่วยในการเพิ่มประสิทธิภาพในการเข้าถึงให้ดีมากขึ้นอีกด้วย

2.6.1.3 Sequence File

Sequence file เป็น service ของ Hadoop อีกวิธีหนึ่งที่ใช้ในการจัดเก็บไฟล์ภายใน HDFS โดยที่ filename จะถูกใช้งานเป็น key และ contents จะเป็น value โดยในการใช้งานนั้นจะต้องทำการเขียน program ขึ้นมาเพื่อแปลงให้ไฟล์นั้นเรียงต่อกัน จากนั้นใช้ Map Reduce technique ในการทำให้เป็น sequence file ในปัจจุบันมีการใช้ sequence file เพื่อจัดเก็บ email เก่าซึ่งเป็นไฟล์ขนาดเล็กจำนวนมากเอาไว้ใน HDFS [5] โดยในการทำงานนั้นจะทำการแปลงไฟล์ให้มีลักษณะเป็น key-value pair และสามารถทำการค้นหา email messages ได้โดยใช้ Hadoop platform แต่ในการค้นหาไฟล์ภายใน sequence file นั้นจะต้องทำการเริ่มหาจากต้นไฟล์เสมอ จึงทำให้มีผลกระทบกับประสิทธิภาพในการเข้าถึงไฟล์ รวมถึงเวลาที่ใช้ในการแปลงไฟล์ให้เป็น sequence file นั้นยังใช้เวลานานอีกด้วย

2.6.2 Multiple NameNode

วิธีในการแก้ปัญหาไฟล์ขนาดเล็กบน Hadoop อีกวิธีคือการขยายความสามารถในการรับ load ของ NameNode ให้สามารถรับ load ได้มากขึ้น ซึ่งสถาปัตยกรรมของ HDFS นั้น ทุกการทำงานจะต้องผ่าน NameNode ซึ่งเป็น master เพียงตัวเดียวเสมอ ดังนั้นหากต้องการเพิ่มความสามารถในการรับ load จะต้องทำการขยาย NameNode ให้มีมากกว่า 1 ตัว

2.6.2.1 HDFS Federation

HDFS Federation [7] นั้นจะทำการเพิ่มจำนวนของ NameNode ให้มากขึ้น โดยที่ NameNode แต่ละตัวนั้นจะมีอิสระจากกันและไม่ต้องทำงานประสานงานกัน ซึ่ง DataNode จะถูกใช้งานเหมือนเป็น storage ทั่วไปที่ถูกใช้งานจาก NameNode ทั้งหมด โดยแต่ละ DataNode จะต้องทำการ register กับ NameNode ทั้งหมดใน cluster และต้องส่ง heartbeat และ block report ให้กับ NameNode ทุกตัวรวมถึงคอยรับคำสั่งจากทุก NameNode ด้วย แต่ปัญหาคือต้องทำการแก้ไขระบบภายในให้รองรับการทำงานแบบที่มี master หลายตัว ซึ่งเมื่อระบบมี master หลายตัวจึงมีความยุ่งยากในการ configure ระบบให้ master ที่มีหลายตัวทำงานร่วมกันได้อย่างสอดคล้อง และไม่มีข้อผิดพลาด

2.6.2.2 HAR file

Apache Hadoop Foundation ได้ทำการพัฒนา HAR ขึ้นมาเหมือนเป็น service ตัวหนึ่งที่อยู่ภายใน HDFS ซึ่งถูกใช้งานเพื่อแก้ปัญหาไฟล์ขนาดเล็กบน HDFS โดย Mackey et al [14] นำ Hadoop archive (HAR) มาใช้เพื่อลดขนาดของ metadata storage requirement ของไฟล์ขนาดเล็กสำหรับ scientific application และสร้างเครื่องมือที่ทำการแจ้งเตือน percent การใช้งาน directory quota ของแต่ละ user กล่าวคือเมื่อ user ทำการเพิ่มไฟล์เข้าไปยัง HDFS directory เครื่องมือนี้ก็จะแจ้งเตือน user ว่าได้ใช้งานไปแล้วกี่ percent เพื่อให้ user ทราบว่า data ที่มีอยู่ในปัจจุบันนั้นกำลังจะเกิน limit ที่สามารถใช้งานได้แล้ว แต่การใช้งาน HAR นั้นเป็นการใช้เพื่อลดขนาด metadata storage เพียงอย่างเดียวเท่านั้น และเป็นการเพิ่ม overhead ที่ไม่จำเป็น จึงทำให้ประสิทธิภาพในการเข้าถึงช้ากว่าการเข้าถึงไฟล์ใน HDFS ปกติ

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

บทที่ 3
แนวคิดและวิธีดำเนินงานวิจัย

ในบทนี้ผู้วิจัยจะอธิบายถึงปัญหาที่เกิดขึ้นของไฟล์ขนาดเล็กบน Hadoop และจะนำเสนอแนวคิดในการแก้ปัญหาดังกล่าว

3.1 ปัญหาไฟล์ขนาดเล็กบน Hadoop

เนื่องจาก HDFS นั้นถูกออกแบบมาเพื่อใช้สำหรับจัดเก็บไฟล์ขนาดใหญ่ เช่นขนาด 1GB, 1TB, 1PB เป็นต้น ดังนั้นเมื่อต้องรับมือกลับไฟล์ที่มีขนาดเล็กที่มีขนาดต่ำว่า 128MB หรือขนาดเล็กกว่าขนาด block ของ HDFS จะทำให้เกิดปัญหาขึ้น โดยสามารถอธิบายปัญหาที่เกิดขึ้นบน HDFS ได้ดังต่อไปนี้

3.1.1 ข้อจำกัดของพื้นที่จัดเก็บบน NameNode

เพื่อให้ระบบมีประสิทธิภาพมากที่สุด NameNode จึงจัดเก็บ file system metadata ทั้งหมดเอาไว้ใน main memory โดยที่ NameNode จะทำการจัดเก็บ ไฟล์, directory และblock แยกกันเสมือนเป็น 1 object ซึ่งหากระบบกำหนดให้ replica เป็น 3 จะทำให้ metadata ของแต่ละ object ต้องใช้พื้นที่ในการจัดเก็บดังนี้[22]

- File size: 250 bytes
- Directory size: 290 bytes
- Block size: 368 bytes

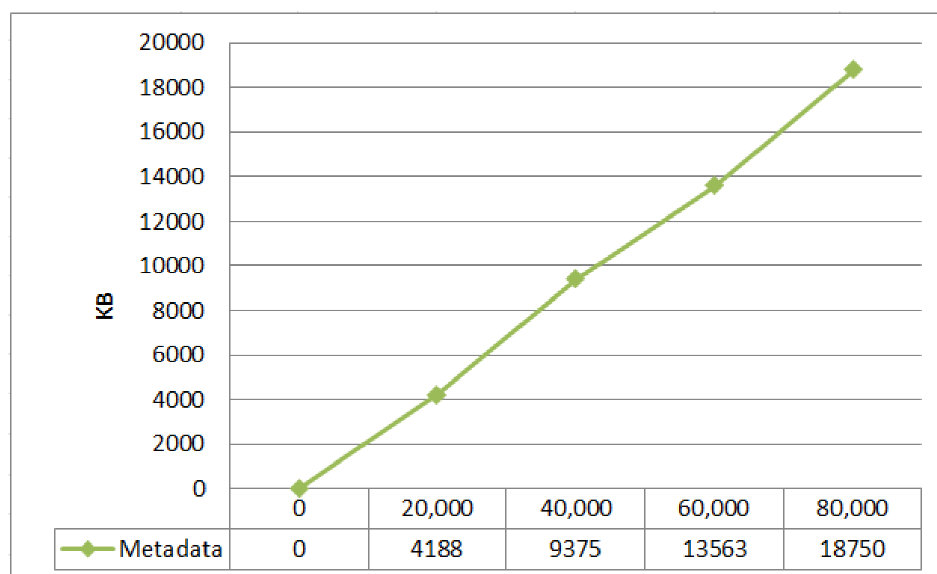
ซึ่งในงานวิจัย “Improving Metadata Management for Small Files in HDFS” [14] ได้มีการทดสอบเพื่อแสดงพื้นที่ที่ใช้ในการจัดเก็บ metadata สำหรับ scientific applications ซึ่งได้ผลลัพธ์ดังตารางที่ 1

ตารางที่ 2 แสดงขนาด metadata ของ scientific application

Application	Number of files	Size of Each File	Dir Meta (Bytes)	File Meta (Bytes)	Block Meta (Bytes)	Total Meta
Climatology	450,000	61MB	7,950	56,250,000	78,918,750	128.91 MB
Biology	20,000,000	1MB	7,950	2,500,000,000	57,500,000	2.38 GB
Astronomy	30,000,000	190KB	7,950	3,750,000,000	16,003,416	3.51 GB

จะเห็นว่าเมื่อทำการจัดเก็บข้อมูลจาก scientific application ทั้ง 3 application จะต้องใช้พื้นที่จัดเก็บบน NameNode เพื่อจัดเก็บ metadata มากถึง 6 GB

โดยในงานวิจัยนี้ได้ทำการทดลองเพื่อประเมินพื้นที่ในการจัดเก็บ metadata บน NameNode เช่นเดียวกัน โดยผลลัพธ์ถูกแสดงอยู่ในภาพที่ 3.1



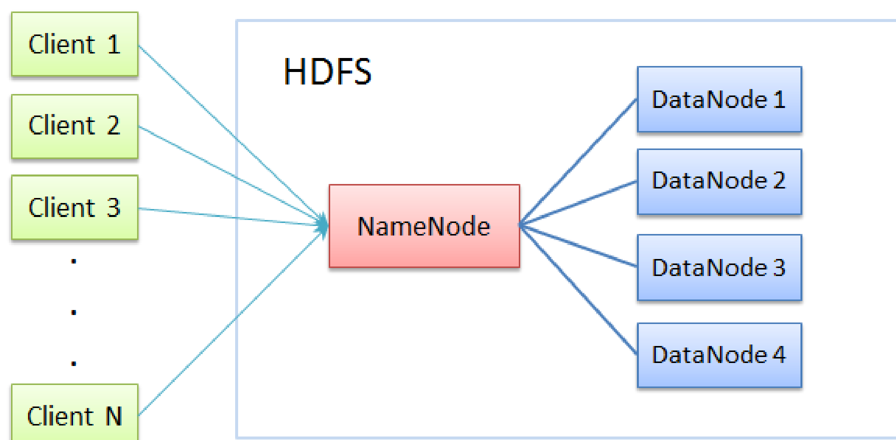
	0	20,000	40,000	60,000	80,000
Metadata	0	4188	9375	13563	18750

ภาพที่ 3.1 กราฟแสดงพื้นที่จัดเก็บ metadata บน NameNode

จะเห็นว่ายิ่งจำนวนไฟล์ขนาดเล็กมีมากเท่าใด ก็ยิ่งต้องใช้พื้นที่ในการจัดเก็บ metadata บน NameNode มากขึ้นเท่านั้น โดยหนึ่งใน production cluster ของ Yahoo! [17] นั้นมีไฟล์ที่มีขนาดเล็กมากถึง 57 ล้านไฟล์ ซึ่งเมื่อทำการจัดเก็บเอาไว้ใน HDFS จะทำให้ต้องใช้พื้นที่จัดเก็บบน NameNode มากถึง 95% แต่ใช้พื้นที่บน disk เพียงแค่ 30% ซึ่งหากทำการเปรียบเทียบกับไฟล์ขนาดใหญ่ในกรณีที่ขนาดโดยรวมเท่ากันนั้น พื้นที่จัดเก็บบน NameNode อาจใช้งานเพียง 30% ทั้งนี้ เนื่องจากพื้นที่บน NameNode มีขีดจำกัด ดังนั้น หากไฟล์ขนาดเล็กบน HDFS มีจำนวนมากเกินไปอาจทำให้พื้นที่บน NameNode เต็มได้

3.1.2 ปัญหาคอขวด (Bottleneck)

การจัดการข้อมูลใน HDFS เป็นงานที่ใช้เวลานานเนื่องจากต้องทำงานร่วมกันระหว่างโหนด ซึ่งในการเข้าถึงไฟล์ ผู้ใช้งานจะต้องทำการร้องขอ metadata จาก NameNode ก่อนทุกครั้ง ดังภาพที่ 3.2



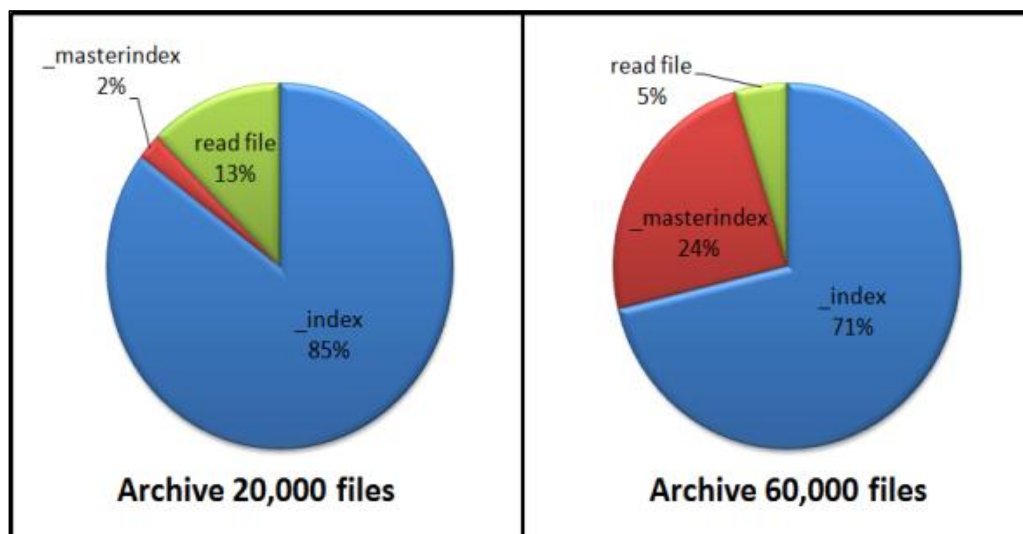
ภาพที่ 3.2 แสดงการเชื่อมต่อไปยัง HDFS

สำหรับไฟล์ขนาดเล็กนั้นการส่งข้อมูล (Data transfer) จะใช้เวลาที่สั้นมากในขณะที่ overhead ส่วนใหญ่จะอยู่ที่ disk seek และการจัดการข้อมูล โดยหากเป็นไฟล์ขนาดใหญ่ ผู้ใช้งานจะต้องทำการเชื่อมต่อไปยัง NameNode เพื่อร้องขอ metadata เพียง 1 ครั้งเท่านั้น ในขณะที่หากเป็นไฟล์ขนาดเล็กในกรณีที่ขนาดไฟล์รวมเท่ากันนั้น ผู้ใช้งานจะต้องทำการเชื่อมต่อไปยัง NameNode หลายครั้งมากยิ่งขึ้นซึ่งจะทำให้มีผลกระทบกับประสิทธิภาพของ NameNode และอาจทำให้เกิดปัญหาคอขวดกับ NameNode ได้

3.1.3 ปัญหาการเข้าถึงไฟล์ภายใน HAR

จากที่ได้กล่าวไปแล้วในหัวข้อที่ 2.2 การใช้งาน HAR นั้นสามารถลดการใช้งาน namespace และลด operation load ของ NameNode ได้อย่างมีประสิทธิภาพ โดยการรวมไฟล์ขนาดเล็กเข้าไว้ด้วยกันให้กลายเป็นไฟล์ขนาดใหญ่ และสร้าง two-level indexes ขึ้นมาเพื่อใช้สำหรับการเข้าถึงไฟล์ แต่ปัญหาคือ การใช้งาน two-level indexes ทำให้เกิด overhead ที่ไม่จำเป็นขึ้นเนื่องจากการอ่านไฟล์จาก HAR ต้องทำการเข้าถึง indexes 2 ครั้งดังแสดงในภาพที่ 3.3

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY



ภาพที่ 3.3 แสดงประสิทธิภาพการเข้าถึงไฟล์ของ HAR

จะเห็นว่าในการเข้าถึง HAR ที่มีไฟล์อยู่ภายใน 20,000 ไฟล์นั้นเวลาส่วนใหญ่จะถูกใช้ในการค้นหาข้อมูลภายใน _index ในขณะที่ _masterindex นั้นใช้เวลาเพียง 2% ของเวลาทั้งหมด แต่เมื่อ HAR มีข้อมูลอยู่ภายในเพิ่มมากขึ้นเป็น 60,000 ไฟล์ เวลาที่ใช้ในการค้นหาข้อมูลภายใน _masterindex นั้นจะเริ่มส่งผลกระทบต่อประสิทธิภาพมากขึ้นอย่างเห็นได้ชัด ซึ่งในที่นี้คือเพิ่มขึ้นมาเป็น 24% ของเวลาที่ใช้ทั้งหมด กล่าวคือยิ่งทำการ archive ไฟล์จำนวนมากขึ้น ก็จะยิ่งส่งผลกระทบต่อประสิทธิภาพการทำงานของ _masterindex

3.2 หลักการทำงานของระบบ NHAR

จากปัญหาที่ได้กล่าวไปแล้วข้างต้น ในงานวิจัยนี้จึงนำเสนอวิธีในการแก้ปัญหาเหล่านั้นโดยนำ Hadoop Archive หรือ HAR มาเป็นพื้นฐานในการแก้ปัญหา โดยแนวคิดพื้นฐานในการแก้ปัญหาในงานวิจัยนี้จะแบ่งออกเป็น 2 ส่วนคือ การรวมไฟล์ที่มีขนาดเล็กเข้าด้วยกันให้กลายเป็นไฟล์ขนาดใหญ่เพื่อลดจำนวนไฟล์ โดยมุ่งเน้นในการเพิ่มประสิทธิภาพในการเข้าถึง และเพิ่มความสามารถในการทำงานของการจัดการไฟล์ที่อยู่ภายใน HAR ให้มีความคล้ายคลึงกับ file system ทั่วไป

3.2.1 นิยามของตัวแปรในระบบ NHAR

ในการอธิบายหลักการทำงานของระบบ NHAR จะใช้นิยามต่อไปนี้

- Ni คือ จำนวน index file

โดยที่ Ni สามารถเปลี่ยนแปลงได้ตามความต้องการของระบบ ซึ่งอาจกำหนดให้มีเพียงแค่ 1 index file ก็ได้ แต่เวลาที่ใช้ในการเข้าถึงอาจช้า ซึ่งหากระบบต้องการให้การเข้าถึงไฟล์มีความรวดเร็ว อาจแบ่งจำนวน index file ให้มีจำนวนเยอะมากขึ้น

- Np คือ จำนวน part file

สืบเนื่องจาก coding ของ HAR ดั้งเดิมนั้นมีการกำหนดให้ขนาดของ part file มีขนาด 2GB เนื่องจากเป็นขนาดของไฟล์ที่เหมาะสมสำหรับการจัดเก็บ ดังนั้นจำนวน part file ที่จะเกิดขึ้นภายใน NHAR จะมีจำนวนดังนี้

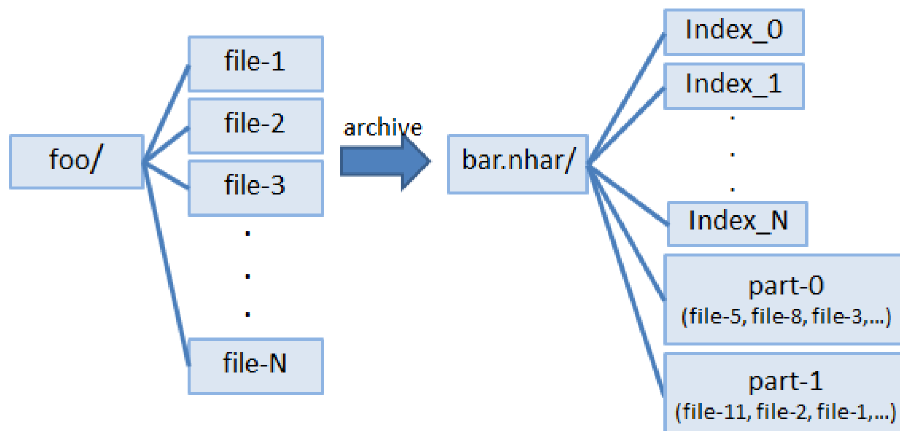
$$\mathrm{Np} = ceil(\frac{\text{Total file size}}{2\ \mathrm{GB}}) \quad (3.1)$$

- Nf คือ จำนวนไฟล์ทั้งหมด
- Fn คือ ชื่อไฟล์
- Fi คือ หมายเลขของ Index file
- Fp คือ หมายเลขของ part file
- Thd คือ เวลาที่ใช้ในการคัดลองไฟล์ 1 ไฟล์จาก local ไปยัง HDFS
- Tnh คือ เวลาที่ใช้ในการเพิ่มไฟล์ 1 ไฟล์เข้าไปยัง NHAR โดยการใช้ Insert Function

3.2.2 ขั้นตอนการสร้าง Archive File ของระบบ NHAR

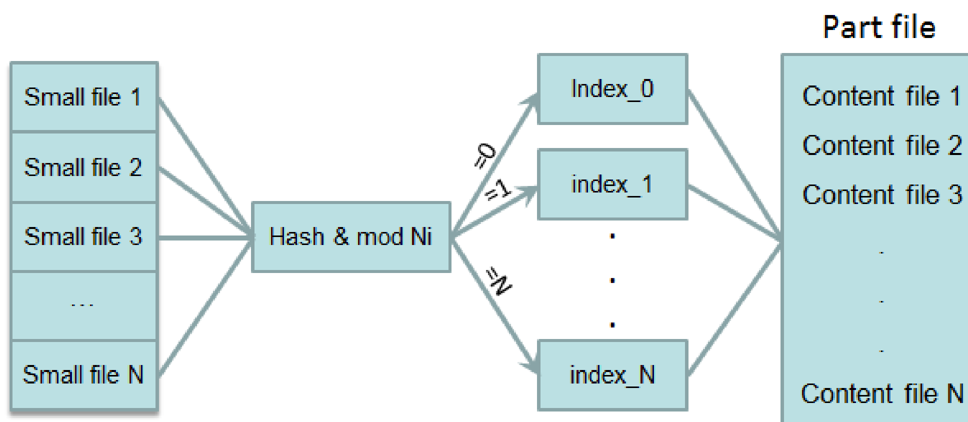
เพื่อเพิ่มประสิทธิภาพในการเข้าถึง ทางผู้วิจัยจึงทำการพัฒนา NHAR ขึ้นมาโดยมีพื้นฐานมาจากโครงสร้างการทำงานของ HAR เดิม แต่ใช้การทำงานของ single-level index แทน โดยเปลี่ยนการใช้งาน master-index เป็นการสร้าง hash table จากข้อมูล index และแบ่งข้อมูลเหล่านั้นให้มีหลาย index ไฟล์แทน ดังที่แสดงในภาพที่ 3.4

จุฬาลงกรณ์มหาวิทยาลัย CHULALONGKORN UNIVERSITY



ภาพที่ 3.4 แสดงโครงสร้างการทำ archive file ของ NHAR

ในการจัดทำ index ของไฟล์นั้น โปรแกรม archive จะนำชื่อไฟล์มาแปลงค่า hash code แล้วทำการ mod ค่า hash code นั้นด้วยจำนวน index file เพื่อดูว่า metadata ของไฟล์นั้นๆควรจะถูกเก็บเอาไว้ที่ index file ใด ในขณะที่ไฟล์ที่แท้จริงจะถูกจัดเก็บเอาไว้ใน Part file เช่นเดียวกับ HAR แบบดั้งเดิม ซึ่งการทำ index file ในลักษณะนี้จะช่วยลดขั้นตอนในการจัดเก็บสถานที่เก็บไฟล์ได้อย่างมีประสิทธิภาพ โดยกลไกในการทำ hashing ของ NHAR จะแสดงอยู่ในภาพที่ 3.5



ภาพที่ 3.5 แสดงเทคนิคในการทำ archive file ของ NHAR

Create(Fn)

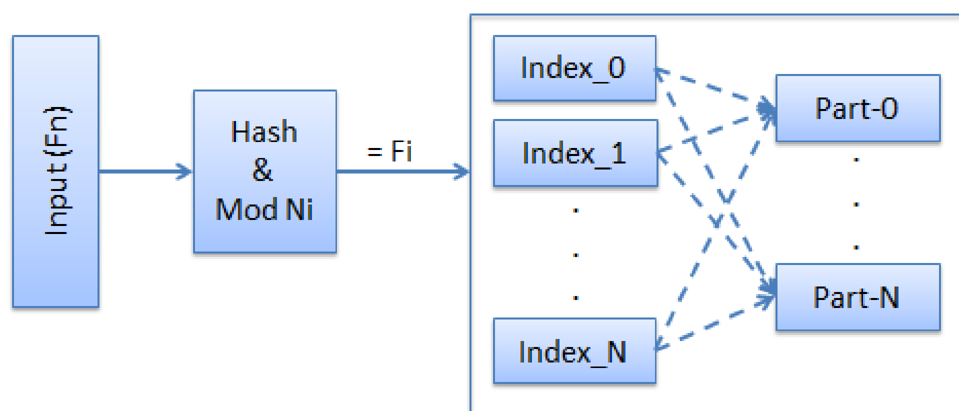
1. นำชื่อไฟล์แต่ละชื่อไฟล์มาทำการเข้ารหัสด้วย Hashing Function แล้วทำการ mod ด้วย Ni ซึ่งเป็นจำนวนของ Index file เพื่อคำนวณหา Fi ซึ่งเป็นหมายเลขของ Index file แล้วจัดเก็บข้อมูล metadata ของไฟล์เอาไว้ใน Index file ดังกล่าวโดยใช้สมการ

$$Fi = hash(Fn) mod\ Ni \qquad (3.2)$$

2. นำข้อมูลของไฟล์จัดเก็บเอาไว้ใน Part file โดยจัดเก็บข้อมูลของแต่ละไฟล์เรียงต่อกัน จน Part file มีขนาด 2GB จึงทำการปิดไฟล์ และจัดเก็บไฟล์ที่เหลือเอาไว้ใน Part file ใหม่ โดยจำนวน Part file จะสามารถหาได้จากสมการที่ 3.1

3.2.3 ขั้นตอนการเข้าถึงไฟล์ใน Archive File ของระบบ NHAR

ในการเข้าถึงไฟล์จะมีกลไกการทำงานเช่นเดียวกับการทำ archive file นั่นคือ จะนำชื่อไฟล์ที่ต้องการค้นหามาทำการแปลงค่า hash code และทำการ mod ค่า hash code เพื่อให้ทราบว่า metadata ของไฟล์นั้นถูกเก็บเอาไว้ใน index file ใด โดยประสิทธิภาพในการเข้าถึงของ NHAR เมื่อเทียบกับ Masterindex ของ HAR นั้นจะเป็น O(1) เนื่องจาก NHAR จะเป็นการทำ hashing ซึ่งจะสามารถเข้าถึง index file ได้เลย ดังแสดงในภาพที่ 3.6



ภาพที่ 3.6 แสดงการเข้าถึงไฟล์ของ NHAR

Access(Fn)

Fn คือ ชื่อไฟล์ที่ต้องการเข้าถึง

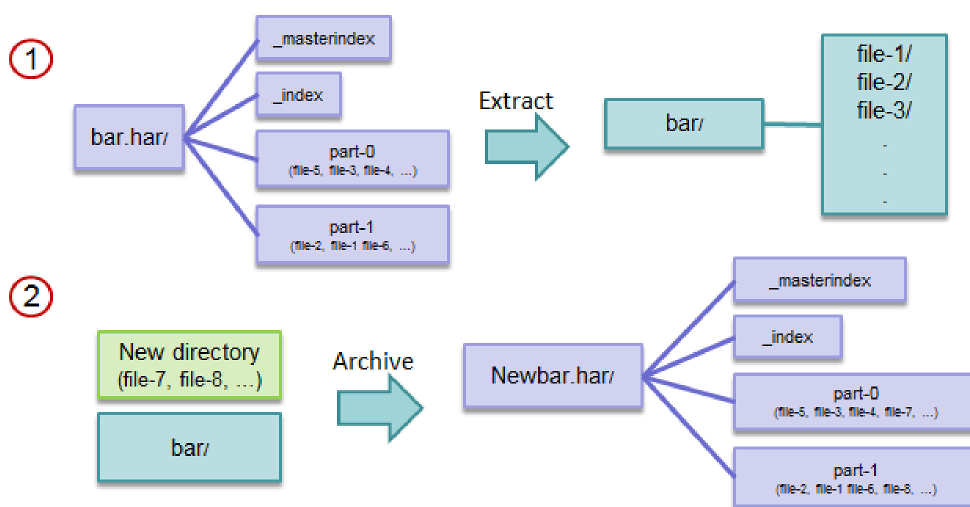
1. นำชื่อของไฟล์มาทำการเข้ารหัสด้วย Hashing Function เพื่อคำนวณหา Fi ซึ่งเป็นหมายเลขของ Index file โดยใช้สมการ

$$\mathrm{Fi} = \mathrm{hash(Fn) mod\ Ni} \quad (3.3)$$

2. ค้นหา metadata ของไฟล์ที่ต้องการภายใน Index file Fi
3. ทำการเข้าถึงข้อมูลไฟล์ที่แท้จริงที่อยู่ภายใน Part file ตามตำแหน่งที่ระบุไว้ใน metadata ของไฟล์

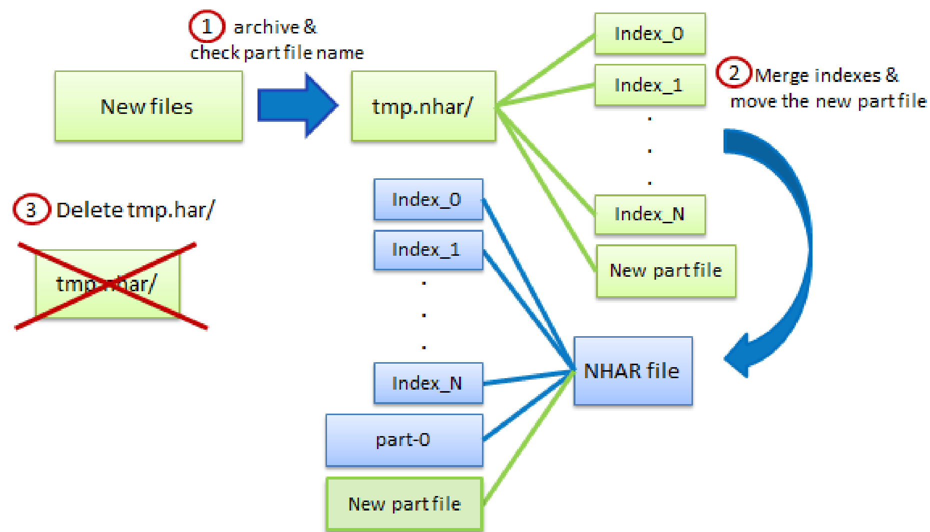
3.2.4 ขั้นตอนการเพิ่มไฟล์ใน Archive File ของระบบ NHAR

ข้อจำกัดอย่างหนึ่งของ HAR คือ เมื่อ HAR file ถูกสร้างขึ้นจะไม่สามารถแก้ไขไฟล์ที่อยู่ภายใน HAR file ได้ เนื่องจาก HAR จะทำการสร้าง hash code ขึ้นมาจาก filename และทำการจัดเก็บเอาไว้ภายใน masterindex file ซึ่งภายใน masterindex จะถูกจัดเรียงตามลำดับของค่า hash code เพื่อให้มีความรวดเร็วในการเข้าถึงไฟล์ ดังนั้นหากต้องการเพิ่มไฟล์เข้าไปใน HAR file ที่ถูกสร้างขึ้นมาแล้วนั้น จำเป็นที่จะต้องค้นหาสถานที่จัดเก็บไฟล์ที่เหมาะสมเพื่อให้เป็นไปตามหลักโครงสร้างเดิมของ HAR ซึ่งทำให้ยากในการจัดการ ดังนั้น HAR จึงไม่อนุญาตให้เพิ่มไฟล์เข้าไปยัง HAR file ที่ถูกสร้างขึ้นมาแล้ว โดยหากต้องการเพิ่มไฟล์เข้าไปยัง HAR file เดิมจะต้องทำการแตกไฟล์ภายในออกมาทั้งหมดก่อน ดังที่แสดงอยู่ในภาพที่ 3.6 ขั้นตอนที่ 1 จากนั้นจึงค่อยทำการสร้าง HAR file ใหม่ขึ้นมาอีกครั้ง โดยรวมไฟล์ที่แตกออกมานั้นกับไฟล์ใหม่ที่ต้องการเพิ่มเข้าไป ดังแสดงอยู่ในภาพที่ 3.7 ขั้นตอนที่ 2 ซึ่งต้องใช้เวลานานและไม่มีประสิทธิภาพเท่าที่ควร



ภาพที่ 3.7 แสดงการเพิ่มไฟล์เข้าไปยัง HAR file ที่มีอยู่แล้วของ HAR

เพื่อแก้ปัญหาในเรื่องข้อจำกัดในการเพิ่มไฟล์เหล่านี้ NHAR จึงยอมให้ user สามารถเพิ่มไฟล์ไปยัง NHAR file ที่มีอยู่แล้วได้โดยไม่ต้องทำการสร้าง NHAR file ใหม่ขึ้นมา ดังที่ได้แสดงอยู่ในภาพที่ 3.8



ภาพที่ 3.8 แสดงการเพิ่มไฟล์เข้าไปยัง NHAR file

กระบวนการในการเพิ่มไฟล์เข้าไปยัง NHAR จะมีกระบวนการหลักอยู่ 3 ขั้นตอนคือ archive file, รวม index files - ย้าย part file ใหม่ และขั้นตอนสุดท้ายคือ ลบ tmp.nhar โดยกระบวนการในการเพิ่มไฟล์จะมีขั้นตอนดังต่อไปนี้

Insert(Fn)

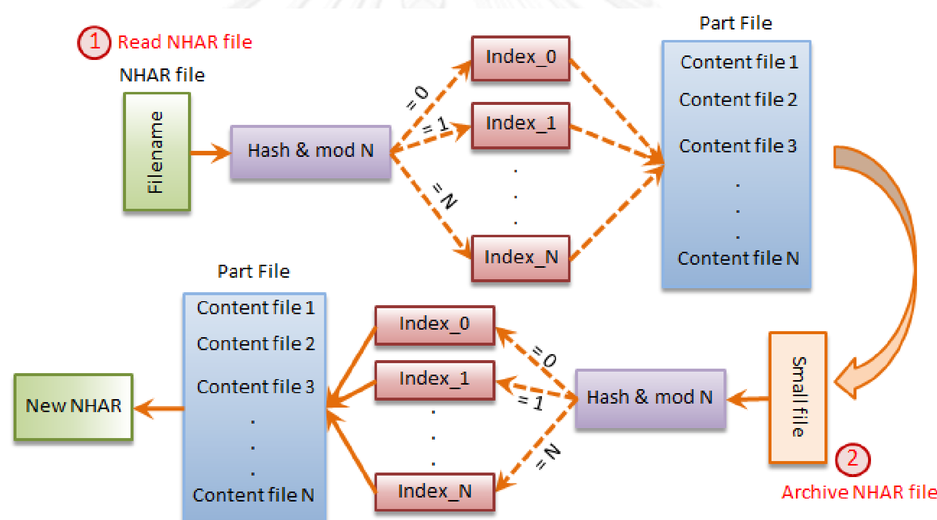
1. ทำการสร้าง archive file ของไฟล์ที่ต้องเพิ่มตามหัวข้อที่ 3.2.3 ขั้นตอนการสร้าง archive file โดยเพิ่มไปยัง tmp.nhar โดยจะทำการตรวจสอบ Fp ซึ่งเป็นหมายเลขของ part file โดยทำการตรวจสอบ Fp ที่อยู่ภายใน NHAR ที่ต้องการเพิ่มไฟล์เข้าไปว่ามีอยู่แล้วหรือไม่เพื่อหลีกเลี่ยงการซ้ำกันของ Fp โดย Fp ที่ถูกอยู่ภายใน tmp.nhar จะสามารถหาได้จากสมการ

$$\mathrm{Fp} = \mathrm{Np(in)} + 1 \qquad (3.4)$$

Np(in) คือ จำนวน part file ที่อยู่ภายใน NHAR ที่ต้องการเพิ่มไฟล์เข้าไป

2. ทำการรวม index file ที่มี Fi เดียวกันใน tmp.nhar กับ index file ที่มีอยู่แล้วใน NHAR file เข้าด้วยกัน และทำการย้าย part Fpทั้งหมดที่อยู่ใน tmp.har ไปยัง NHAR file ที่ต้องการ
3. ลบ tmp.nhar

อย่างไรก็ตาม กระบวนการในการเพิ่มไฟล์เหล่านี้นั้น ในการเพิ่มไฟล์แต่ละครั้งจะเป็นการเพิ่ม part file ใหม่ขึ้นทุกครั้ง ถึงแม้ว่า part file ที่ถูกเพิ่มเข้ามาใหม่นั้นจะมีขนาดเล็กกว่าขนาด default part size ของ HDFS กล่าวคือ ยิ่งใช้งาน inserting function มากเท่าใด ก็ยิ่งต้องใช้พื้นที่ในการจัดเก็บ metadata บน NameNode มากขึ้นเท่านั้น โดยเหตุผลที่ทำให้ระบบไม่สามารถรวม part file เดิมที่มีอยู่แล้วกับ part file ใหม่ได้นั้น เนื่องจากโครงสร้าง index ของ NHAR นั้น จะจัดเก็บสถานที่ของข้อมูลที่แท้จริงที่อยู่ภายใน part file เอาไว้ ดังนั้นหากระบบทำการแก้ไขข้อมูลที่อยู่ภายใน part file จะทำให้สถานที่เก็บข้อมูลเปลี่ยน และต้องทำการสร้างข้อมูล index ขึ้นมาใหม่ ซึ่งก็คือต้องทำการสร้าง archive file ขึ้นมาใหม่ทั้งหมด ดังนั้นในกระบวนการการเพิ่มไฟล์จะทำการตรวจสอบขนาดของ part file ทั้งหมด ว่าขนาดของ part file ทั้งหมดนั้นมีขนาดเกินกว่าขนาดของ block size นั่นคือ 128MB หรือไม่ โดยจะทำการแจ้งเตือนผู้ใช้งานว่าในขณะนี้ part file ใน NHAR มีจำนวนเท่าใด และหากทำการ reorganize (recreating) จะเหลือ part file เท่าใด โดยขั้นตอนในการทำ reorganize จะถูกแสดงอยู่ในภาพที่ 3.9



ภาพที่ 3.9 แสดงการทำ reorganization ของ NHAR

กระบวนการในการทำ reorganization จะประกอบด้วย 2 ขั้นตอนหลักคือการอ่านไฟล์ที่อยู่ภายใน NHAR file แล้วทำการสร้าง NHAR file ใหม่ขึ้นมาโดยจะมีขั้นตอนดังต่อไปนี้

Reorganize(Fn)

1. นำชื่อของไฟล์แต่ละชื่อไฟล์ที่อยู่ภายใน NHAR มาทำการเข้ารหัสด้วย Hashing Function เพื่อคำนวณหา Fi ซึ่งเป็นหมายเลขของ Index file โดยใช้สมการที่ 3.3
2. ค้นหา metadata ของไฟล์ที่ต้องการภายใน Index file Fi
3. ทำการเข้าถึงข้อมูลไฟล์ที่แท้จริงที่อยู่ภายใน Part file ตามตำแหน่งที่ระบุไว้ใน metadata ของไฟล์

4. หลังจากได้ไฟล์แต่ละไฟล์แล้วจะนำไฟล์นั้นไปทำการสร้าง archive file ใหม่ต่อทันทีโดยระบบจะนำชื่อไฟล์นั้นไปทำการเข้ารหัสด้วย Hashing Function แล้วทำการ mod ด้วย Ni ซึ่งเป็นจำนวนของ Index file เพื่อคำนวณหา Fi ซึ่งเป็นหมายเลขของ Index file แล้วจัดเก็บข้อมูล metadata ของไฟล์เอาไว้ใน Index file ดังกล่าวโดยใช้สมการที่ 3.2
5. นำข้อมูลของไฟล์จัดเก็บเอาไว้ใน Part file โดยจัดเก็บข้อมูลของแต่ละไฟล์เรียงต่อกันจน Part file มีขนาด 2GB จึงทำการปิดไฟล์ และจัดเก็บไฟล์ที่เหลือเอาไว้ใน Part file ใหม่ โดยจำนวน Part file จะสามารถหาได้จากสมการที่ 3.1

3.3 ตัวอย่างการทำงานของระบบ

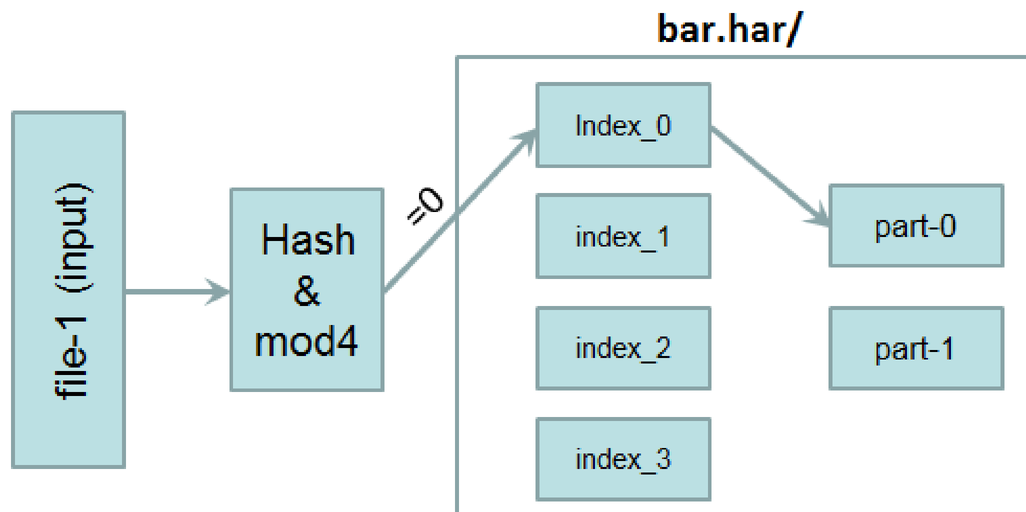
ตัวอย่างในการค้นหาไฟล์ถูกแสดงอยู่ในภาพที่ 3.10 โดยในตัวอย่างนี้เรากำหนดให้ Ni = 4 และเราต้องการหา file-1 ดังนั้นเราสามารถดูสถานที่จัดเก็บไฟล์ได้จาก

Index file storage = hash (file-1) mod 4

= 23699768 mod 4

= 0

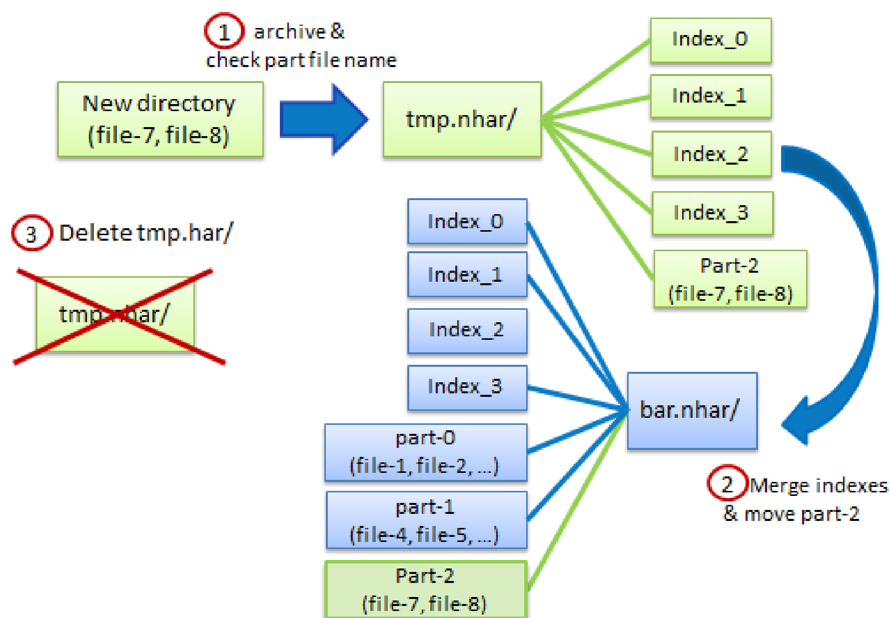
แสดงว่า metadata ของ file-1 ถูกจัดเก็บเอาไว้ใน index_0 ซึ่งเมื่อโปรแกรมทำการค้นหาใน index_0 ก็จะพบสถานที่เก็บไฟล์ที่แท้จริงของ file-1 ที่อยู่ภายใน Part file



ภาพที่ 3.10 แสดงตัวอย่างในการเข้าถึงไฟล์ของ NHAR

3.3.1 ตัวอย่างการเพิ่มไฟล์เข้าไปภายใน NHAR file

ดังที่ได้อธิบายไปแล้วในหัวข้อที่ 3.2.4 โดยหากต้องการเพิ่ม file-7 และ file-8 เข้าไปใน bar.nhar กระบวนการในการเพิ่มไฟล์จะแสดงอยู่ในภาพที่ 3.11



ภาพที่ 3.11 แสดงตัวอย่างในการเพิ่มไฟล์เข้าไปยัง NHAR file

โดยจะมีขั้นตอนดังนี้

1. ทำการ archive file ที่ต้องการเพิ่ม (file-7, file-8) ไปยัง tmp.nhar โดยมีกระบวนการในการทำเช่นเดียวกับโปรแกรม archive ของ NHAR แต่จะมีการตรวจสอบชื่อของ part file (Part-0, Part-1,...) ใน bar.nhar ว่ามีอยู่แล้วหรือไม่เพื่อหลีกเลี่ยงการซ้ำกันของชื่อ part file
2. ทำการรวม index file ที่มีชื่อเดียวกันใน tmp.nhar กับ index file ที่มีอยู่แล้วใน bar.nhar เข้าด้วยกัน แล้วทำการย้าย part file ทั้งหมดที่อยู่ใน tmp.nhar ไปยัง bar.nhar
3. ลบ tmp.nhar

จากภาพที่ 3.11 ก่อนทำการเพิ่มไฟล์นั้น bar.nhar จะประกอบด้วย 2 part file คือ part-0 และ part-1 เมื่อเราทำการรัน inserting function จะทำให้เกิด part file ใหม่ขึ้นมาคือ part-2 โดยเราสามารถหาหาขนาดของ metadata ทั้งหมดหลังจากเพิ่มไฟล์ได้ดังนี้

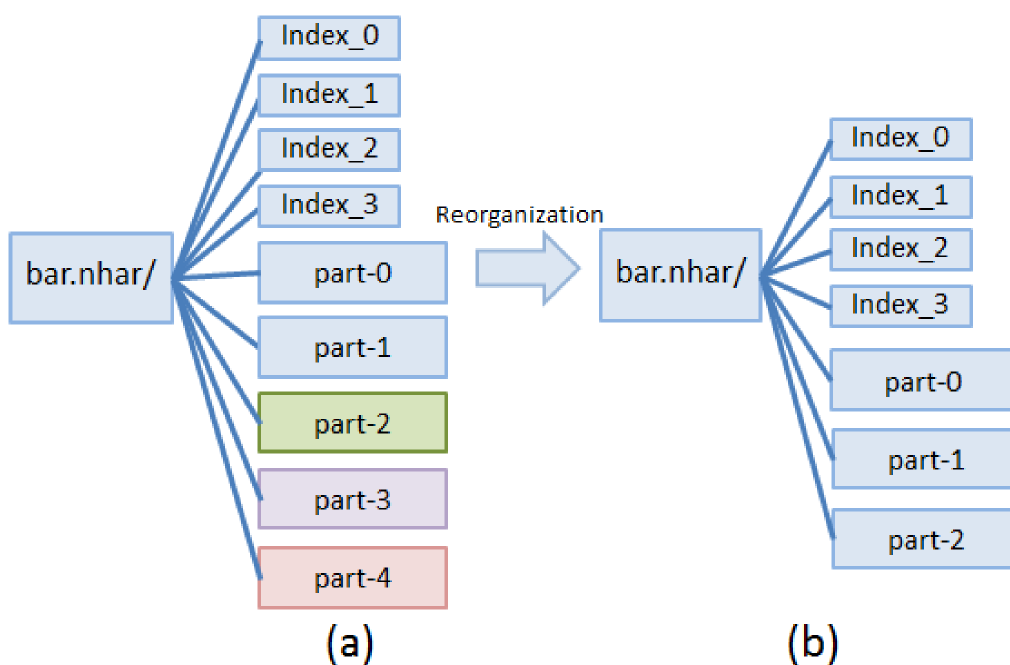
Total metadata = (Ni + Np)*250

= (4 + 3) *250

= 7*250

= 1750 bytes

จะเห็นว่าเมื่อทำการเพิ่มไฟล์จะเกิดไฟล์ใหม่ขึ้นมาและต้องใช้พื้นที่ในการจัดเก็บ metadata เพิ่มมากขึ้น โดยเมื่อทำการรัน inserting function อีกครั้ง ก็จะเกิด part file ใหม่เพิ่มขึ้นมาอีกเรื่อยๆ ซึ่งไฟล์ที่เกิดขึ้นใหม่นั้นเป็นไฟล์ขนาดเล็กและเมื่อเพิ่มไฟล์หลายครั้งก็จะทำให้เกิดปัญหาขึ้นได้ โดยในภาพที่ 3.12 จะแสดงตัวอย่างโครงสร้างที่อยู่ภายใน NHAR ก่อนและหลังจากทำการ reorganize แล้ว



ภาพที่ 3.12 (a) แสดงโครงสร้างของไฟล์หลังจากใช้งาน inserting function
(b) แสดงโครงสร้างของไฟล์หลังจากทำการ reorganization

ภาพที่ 3.12(a) แสดงโครงสร้างของไฟล์หลังจากใช้งาน inserting function โดยที่ part-0 และ part-1 เป็นไฟล์เดิมที่มีอยู่แล้วภายใน bar.nhar ก่อนที่จะใช้งาน inserting function ส่วน part-2, part-3 และ part-4 เป็นไฟล์ที่เกิดขึ้นจากการใช้งาน inserting function 3 ครั้ง ซึ่งในกรณีที่ part-0 มีขนาดเท่ากับ block size ส่วน part-1, part-2, part-3 และ part-4 มีขนาดน้อยกว่า block size

ภาพที่ 3.12(b) แสดงโครงสร้างของไฟล์หลังจากกระบวนการ reorganization โดยในกรณีที่ part-1, part-2, part-3 และ part-4 มีขนาดน้อยกว่า block size เมื่อรัน reorganizing function

แล้ว part file เหล่านี้ก็จะถูกรวมเข้าด้วยกันโดยจะถูกแบ่งตามขนาดของ block size ซึ่งจะทำให้ลดการใช้งาน memory บน NameNode ได้

โดยเราสามารถคำนวณพื้นที่เก็บ metadata ของภาพที่ 3.10(a) ซึ่งเป็นการเพิ่มไฟล์ 3 ครั้ง ได้ดังนี้

Total metadata = (Ni + Np)*250

= (4 + 5) *250

= 9 * 250

= 2250 bytes

ในขณะที่หากเราทำการ reorganize จะเหลือพื้นที่มีการจัดเก็บ metadata ดังนี้

Total metadata $= \left(\mathrm{Ni} + ceil\left(\frac{\text{total new files size}}{2\text{ GB}}\right)\right) * 250$

$= \left(4 + ceil\left(\frac{5\text{ GB}}{2\text{ GB}}\right)\right) * 250$

= (4+3) * 250

= 1750 bytes

จะเห็นว่าเมื่อทำการ reorganize จะสามารถลดการใช้งานบน NameNode ได้

3.4 เปรียบเทียบการเพิ่มไฟล์เข้ายัง HDFS และ NHAR

โดยปกติแล้ว หากผู้ใช้งานต้องการใช้งาน Hadoop จะต้องทำการคัดลอกไฟล์ที่อยู่ภายใน local เข้าไปยัง HDFS ซึ่งผู้ใช้สามารถคัดลอกไฟล์ได้เหมือน file system ทั่วไป โดยใช้คำสั่งที่ Hadoop จัดเตรียมไว้ให้ ซึ่งเวลาที่ใช้ในการคัดลอกไฟล์จะขึ้นอยู่กับขนาดของไฟล์ที่ทำการคัดลอก ซึ่งเวลาเฉลี่ยที่ใช้ในการคัดลอกไฟล์ 1 ไฟล์ที่มีขนาดอยู่ระหว่าง 1MB – 200 MB คือ Thd = 4 วินาที

สำหรับ NHAR หากผู้ใช้งานต้องการเพิ่มไฟล์เข้าไปยัง NHAR โดยใช้ Insert function นั้น ผู้ใช้ต้องทำการคัดลอกไฟล์ที่ต้องการเพิ่มเข้าไปยัง HDFS ก่อน จากนั้นจึงค่อยเรียกใช้งาน Insert function เพื่อเพิ่มไฟล์นั้นเข้าไปยัง NHAR โดยที่เวลาเฉลี่ยที่ใช้ในการเพิ่มไฟล์ 1 ไฟล์เข้าไปยัง NHAR คือ Tnh = 10 วินาที ดังนั้นเวลาเฉลี่ยที่ใช้ในการเพิ่มไฟล์เข้าไปยัง NHAR คือ Thd + Tnh = 4+10 = 14 วินาที (14-4)/4 = 10/4 = 2.5 กล่าวคือ ในการใช้งาน Insert function นั้นจะต้องใช้เวลามากกว่าการคัดลอกไฟล์เข้าไปยัง HDFS โดยตรงประมาณ 2.5 เท่า แต่ถึงอย่างไรก็ตาม หากทำการเพิ่มไฟล์เข้าไปยัง HDFS โดยตรงจำนวนมากจะทำให้มีปัญหาในเรื่องพื้นที่จัดเก็บบน NameNode ตามที่ได้กล่าวไปแล้วข้างต้น ซึ่งการใช้งาน Insert function นั้นจะช่วยลดในเรื่องพื้นที่จัดเก็บบน NameNode ได้อย่างมีประสิทธิภาพ

บทที่ 4
การพัฒนาเครื่องมือและการทดลอง

ในบทนี้จะเป็นการอธิบายองค์ประกอบของการพัฒนาเครื่องมือที่ใช้เพื่อวิเคราะห์ประสิทธิภาพในการทำงานของระบบ โดยเริ่มจากสภาพแวดล้อมที่ใช้ในการพัฒนาเครื่องมือ จากนั้นอธิบายวิธีการทดลองต่างๆเพื่อวัดประสิทธิภาพในการทำงานภายในระบบดังที่นำเสนอในบทที่ 3 รวมถึงทำการเปรียบเทียบประสิทธิภาพของ NHAR กับ HDFS และ HAR อีกด้วย ซึ่งมีรายละเอียดดังต่อไปนี้

4.1 สภาพแวดล้อมที่ใช้ในการทดลอง

ในงานวิจัยนี้จะทำการสร้าง cluster ขึ้นมาจำนวน 5 node เพื่อใช้เป็น platform สำหรับทำการทดสอบ โดยกำหนดให้ 1 node ทำหน้าที่เป็น NameNode และอีก 4 node ที่เหลือรับหน้าที่เป็น DataNode โดยที่ HDFS จะถูกกำหนดให้จำนวนการทำสำเนา (Replica) เป็น 3 ไฟล์ และบล็อกของ HDFS มีขนาดเป็น 128 MB โดยสภาพแวดล้อมที่ใช้ในการพัฒนาเครื่องมือมีรายละเอียดดังต่อไปนี้

1. ฮาร์ดแวร์ (Hardware)
   - 1.1 หน่วยประมวลผลอินเทลคอร์ไอไฟว์-2500 3.30 กิกะเฮิร์ท (Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz Processors)
   - 1.2 หน่วยความจำสำรอง (Ram) 4 GB
2. ซอฟต์แวร์ (Software)
   - 2.1 ระบบปฏิบัติการ (Operating System) GNU/Linux 3.0.0-26-server x86_64
   - 2.2 Hadoop version 0.20.2-cdh3u5
   - 2.3 HAR File System version 0.20.2-cdh3u5
   - 2.4 Java version 1.6.0_24
3. ข้อมูล (Data)
   - 3.1 ประเภทของไฟล์ เป็น Text file และ system log file
   - 3.2 แต่ละไฟล์จะมีขนาดอยู่ระหว่าง 2 KB ถึง 40 MB
   - 3.3 แบ่งไฟล์ออกเป็นกลุ่มเพื่อทำการทดสอบประสิทธิภาพ โดยแบ่งตามจำนวนไฟล์ออกเป็น 20 000, 40 000, 60 000 และ 80 000 ไฟล์ โดยมีขนาดไฟล์รวมเป็น 1.4 GB, 2.8 GB, 4 GB และ 5.6 GB ตามลำดับ

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

4.2 การพัฒนาเครื่องมือ

ในส่วนของการพัฒนาเครื่องมือสำหรับงานวิจัยนี้ จะเป็นการแสดงผลลัพธ์ในการทำงานของเครื่องมือที่ถูกพัฒนาเพื่อใช้งานสำหรับงานวิจัยนี้ดังที่ได้อธิบายไปแล้วในบทที่ 3

4.2.1 ตัวอย่างโครงสร้างของ NHAR

จากที่ได้กล่าวไปแล้วในบทที่ 3 ทางผู้วิจัยทำการพัฒนา NHAR ขึ้นมาซึ่งทำการปรับเปลี่ยนโครงสร้างการทำ index ของ HAR ดั้งเดิมโดยตัด _masterindex ออกและทำการการสร้าง hash table จากข้อมูล index และแบ่งข้อมูลของ index ออกจากกันเพื่อให้มีหลาย index ไฟล์แทน ภาพที่ 4.1 แสดงตัวอย่างการสร้าง archive file ของ NHAR

```
13/09/09 10:26:37 INFO mapred.JobClient:  map 17% reduce 0%
13/09/09 10:26:41 INFO mapred.JobClient:  map 21% reduce 0%
13/09/09 10:26:48 INFO mapred.JobClient:  map 23% reduce 0%
13/09/09 10:26:57 INFO mapred.JobClient:  map 24% reduce 0%
13/09/09 10:27:01 INFO mapred.JobClient:  map 28% reduce 0%
13/09/09 10:27:03 INFO mapred.JobClient:  map 29% reduce 0%
13/09/09 10:27:06 INFO mapred.JobClient:  map 32% reduce 0%
13/09/09 10:27:09 INFO mapred.JobClient:  map 33% reduce 0%
13/09/09 10:27:12 INFO mapred.JobClient:  map 34% reduce 0%
13/09/09 10:27:19 INFO mapred.JobClient:  map 35% reduce 0%
13/09/09 10:27:39 INFO mapred.JobClient:  map 36% reduce 0%
13/09/09 10:27:43 INFO mapred.JobClient:  map 39% reduce 0%
13/09/09 10:27:46 INFO mapred.JobClient:  map 41% reduce 0%
13/09/09 10:27:49 INFO mapred.JobClient:  map 43% reduce 0%
13/09/09 10:27:52 INFO mapred.JobClient:  map 45% reduce 0%
13/09/09 10:27:55 INFO mapred.JobClient:  map 48% reduce 0%
13/09/09 10:27:58 INFO mapred.JobClient:  map 51% reduce 0%
13/09/09 10:28:01 INFO mapred.JobClient:  map 55% reduce 0%
13/09/09 10:28:04 INFO mapred.JobClient:  map 56% reduce 0%
13/09/09 10:28:07 INFO mapred.JobClient:  map 58% reduce 0%
13/09/09 10:28:10 INFO mapred.JobClient:  map 60% reduce 0%
13/09/09 10:28:13 INFO mapred.JobClient:  map 64% reduce 0%
```

ภาพที่ 4.1 แสดงตัวอย่างหน้าจอในระหว่างการสร้าง archive file ของ NHAR

ภาพที่ 4.2 แสดงข้อมูลที่อยู่ภายใน NHAR หลังจากการใช้งานเครื่องมือ archive เพื่อทำ archive ไฟล์จำนวน 2 หมื่นไฟล์

```
panya@hadoop-master:~/hadoop$ bin/hadoop -ls /user/panya/NHAR/20K.nhar
Found 5 items
-rw-r--r--   1 panya supergroup     946490 2014-07-16 17:04 /user/panya/NHAR/20K
.nhar/index_0
-rw-r--r--   1 panya supergroup     942710 2014-07-16 17:04 /user/panya/NHAR/20K
.nhar/index_1
-rw-r--r--   1 panya supergroup     921440 2014-07-16 17:04 /user/panya/NHAR/20K
.nhar/index_2
-rw-r--r--   1 panya supergroup     871730 2014-07-16 17:04 /user/panya/NHAR/20K
.nhar/index_3
-rw-r--r-- _ 1 panya supergroup  853647275 2014-07-16 17:05 /user/panya/NHAR/20K
.nhar/part-0
```

ภาพที่ 4.2 แสดงตัวอย่างโครงสร้างข้อมูลที่อยู่ภายใน NHAR directory

ซึ่งข้อมูล NHAR ที่อยู่ภายใน directory ของ HDFS หลังจากทำการรันเครื่องมือเพื่อใช้ในการทำ NHAR file นั้น HDFS จะเห็นข้อมูลเพียงแค่ 5 ไฟล์เท่านั้น ซึ่งจะทำให้พื้นที่ที่ใช้ในการจัดเก็บ metadata บน NameNode นั้นลดลงอย่างมาก โดยตัวอย่างข้อมูลที่อยู่ภายใน index file จะแสดงอยู่ในภาพที่ 4.3

```
/sys6/syslogALL/syslog-20120510/syslogALL20120510-055700267+0700.1695134770248129.00000020 file part-0 2428773018 248193 136107617533 7+420+panya+supergroup
/sys4/syslogALL/syslog-20120606/syslogALL20120606-222001271+0700.127010031513985.00000023 file part-0 1599653070 2566 1361038450091+420+panya+supergroup
/sys2/sys/syslogALL/syslog-20120529/syslogALL20120529-215954473+0700.1142492507213549.00001914 file part-0 732782901 434 1361012846764+420+panya+supergroup
/testHAR8/cinderella00dalziala_djvu.txt file part-1 2722732447 16721 1361118504101+420+panya+supergroup
/test/memoirsofsherloc00doyliala_djvu.txt file part-1 824805022 516916 1361028444944+420+panya+supergroup
/sys4/syslogALL/syslog-20120509/syslogALL20120509-175729335+0700.1651963838332430.00000020 file part-0 1383233175 248193 1361036958329+420+panya+supergroup
/sys2/sys/syslogALL/syslog-20120526/syslogALL20120526-161709822+0700.862727856132925.00001914 file part-0 732440830 420 1361012825198+420+panya+supergroup
/sys6/syslogALL/syslog-20120607/syslogALL20120607-020510174+0700.140518933935577.00000023 file part-0 2465607065 2564 1361077507544+420+panya+supergroup
/sys3/syslogALL/syslog-20120504/syslogALL20120504-061450862+0700.1177805364500468.00000021 file part-0 895280547 714510 1361029859951+420+panya+supergroup
/sys3/syslogALL/syslog-20120512/syslogALL20120512-014927790+0700.1853082293044899.00000020 file part-0 1157424965 18107 1361031289143+420+panya+supergroup
/sys6/syslogALL/syslog-20120512/syslogALL20120512-050543181+0700.1864857683344973.00000020 file part-0 2459717661 18107 1361076718021+420+panya+supergroup
/sys6/syslogALL/syslog-20120505/syslogALL20120505-081632115+0700.1271506618251367.00000021 file part-0 2199763236 422 1361074527070+420+panya+supergroup
/sys/syslogALL/syslog-20120519/syslogALL20120519-095958078+0700.235296112300389.00000020 file part-0 297281150 432 1353993796481+420+panya+supergroup
/sys6/syslogALL/syslog-20120509/syslogALL20120509-151817749+0700.1642412251514712.00000020 file part-0 2209581642 248193 1361074850396+420+panya+supergroup
/sys2/sys/syslogALL/syslog-20120504/syslogALL20120504-053849552+0700.1175644054644717.00000021 file part-0 455940624 714510 1361012128074+420+panya+supergroup
/sys7/sysKankrow/syslog20120407-161936931+0700.2776472382283976.00000034 file part-0 2599170856 1025494 1361113699883+420+panya+supergroup
/sys3/syslogALL/syslog-20120607/syslogALL20120607-123005051+0700.178013811028340.00000023 file part-0 1167254363 2567 1361031544329+420+panya+supergroup
/sys/syslogALL/syslog-20120503/syslogALL20120503-130010021+0700.1115724523465828.00000021 file part-0 8686466 2980 1353993667135+420+panya+supergroup
```

ภาพที่ 4.3 แสดงตัวอย่างข้อมูลที่อยู่ภายใน index file

โครงสร้างข้อมูลที่ถูกจัดเก็บเอาไว้ภายใน index file จะมีลักษณะข้อมูลเช่นเดียวกับ index file ของ HAR แบบดั้งเดิม ซึ่งจะประกอบด้วย ชื่อไฟล์, ลักษณะของข้อมูล (file, dir), part file ที่จัดเก็บไฟล์นั้นๆ, ตำแหน่งเริ่มต้นของไฟล์ (startPos), ความยาวของไฟล์ (Length), คุณลักษณะของไฟล์ (Propreties) โดยไฟล์จะถูกจัดเก็บแยกตามผลลัพท์ของการ mod ค่า hash

```
13/12/23 09:06:16 INFO mapred.JobClient:     Map input bytes=217
13/12/23 09:06:16 INFO mapred.JobClient:     Combine input records=0
13/12/23 09:06:16 INFO mapred.JobClient:     SPLIT_RAW_BYTES=145
13/12/23 09:06:16 INFO mapred.JobClient:     Reduce input records=4
13/12/23 09:06:16 INFO mapred.JobClient:     Reduce input groups=4
13/12/23 09:06:16 INFO mapred.JobClient:     Combine output records=0
14/12/23 09:06:16 INFO mapred.JobClient:     Physical memory (bytes) snapshot=26
0890624
13/12/23 09:06:16 INFO mapred.JobClient:     Reduce output records=0
13/12/23 09:06:16 INFO mapred.JobClient:     Virtual memory (bytes) snapshot=212
1273344
13/12/23 09:06:16 INFO mapred.JobClient:     Map output records=4
Inserting files completed
The number of Part file is 3
If reorganize, they will be reduce to 2 part files
panya@hadoop-master:~$ _
```

ภาพที่ 4.4 แสดงตัวอย่างการเพิ่มไฟล์

จากที่ได้กล่าวไปแล้วในหัวข้อที่ 3.2.2 ในการเพิ่มไฟล์นั้น part file จะถูกเพิ่มขึ้นทุกครั้งที่ทำการใช้งาน ดังนั้นในการเพิ่มไฟล์โปรแกรมจะมีการแจ้งเตือนผู้ใช้ว่าตอนนี้มี part file อยู่ทั้งหมดกี่ part file และหากทำการ reorganize จะลดลงเหลือกี่ไฟล์ดังที่ด้แสดงอยู่ในภาพที่ 4.4 โดยโครงสร้างไฟล์หลังจากทำการเพิ่มไฟล์แล้วจะแสดงอยู่ในภาพที่ 4.5

```
panya@hadoop-master:~/hadoop$ bin/hadoop -ls /user/panya/NHAR/New64M.nhar
Found 7 items
-rw-r--r--   1 panya supergroup    2646490 2014-07-16 18:27 /user/panya/NHAR/New
64M.nhar/index_0
-rw-r--r--   1 panya supergroup    2438299 2014-07-16 18:27 /user/panya/NHAR/New
64M.nhar/index_1
-rw-r--r--   1 panya supergroup    1984385 2014-07-16 18:27 /user/panya/NHAR/New
64M.nhar/index_2
-rw-r--r--   1 panya supergroup    2521440 2014-07-16 18:27 /user/panya/NHAR/New
64M.nhar/index_3
-rw-r--r--   1 panya supergroup 2147483648 2014-07-16 17:50 /user/panya/NHAR/New
64M.nhar/part-0
-rw-r--r--   1 panya supergroup   53647275 2014-07-16 17:51 /user/panya/NHAR/New
64M.nhar/part-1
-rw-r--r--   1 panya supergroup    3796033 2014-07-16 18:28 /user/panya/NHAR/New
64M.nhar/part-2
panya@hadoop-master:~/hadoop$ _
```

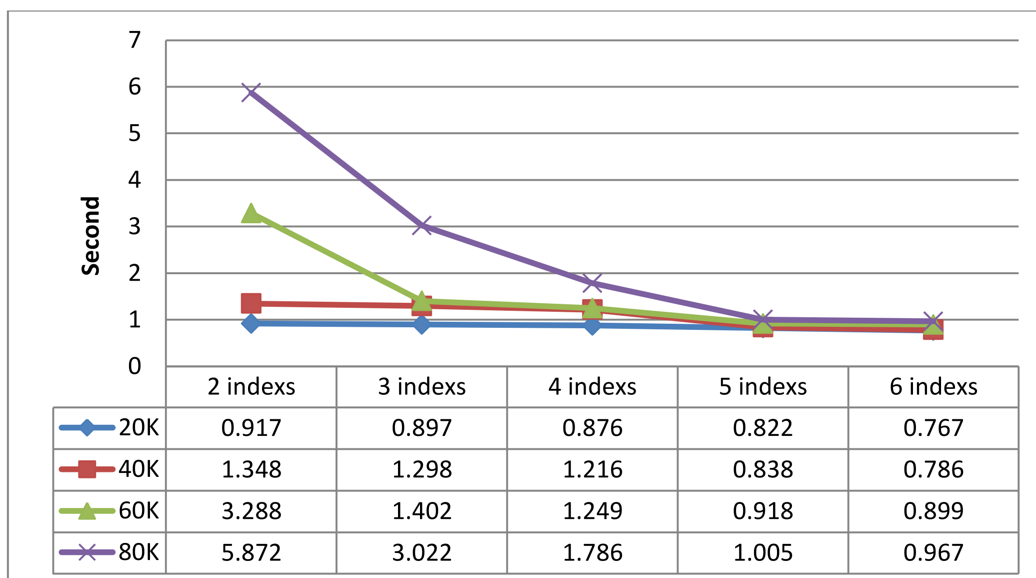
ภาพที่ 4.5 แสดงตัวอย่างโครงสร้างไฟล์หลังจากเพิ่มไฟล์

4.3 การทดลอง

ในส่วนของการทดลองสำหรับงานวิจัยนี้จะทำการทดลองเพื่อทดสอบประสิทธิภาพของเครื่องมือ โดยแบ่งการทดลองออกเป็น 4 การทดลองหลักคือ การทดลองการใช้งานพื้นที่จัดเก็บบน NameNode (Memory usage), การทดลองเวลาที่ใช้ในการสร้าง NHAR file, การทดลองเวลาที่ใช้ในการอ่านไฟล์ภายใน NHAR และ การทดลองการเพิ่มไฟล์เข้าไปยัง NHAR file

นอกจากนั้นแล้ว ทางผู้วิจัยยังทำการทดลองเพื่อเพิ่มประสิทธิภาพของ NHAR โดยทำการทดลองเพื่อให้ทราบถึงขีดจำกัดความสามารถในการจัดเก็บไฟล์ของ index file อีกด้วย ดังแสดงอยู่ในภาพที่ 4.6

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY



	2 indexs	3 indexs	4 indexs	5 indexs	6 indexs
20K	0.917	0.897	0.876	0.822	0.767
40K	1.348	1.298	1.216	0.838	0.786
60K	3.288	1.402	1.249	0.918	0.899
80K	5.872	3.022	1.786	1.005	0.967

ภาพที่ 4.6 กราฟผลการทดลองการวัดประสิทธิภาพโดยแยกตามจำนวน index file

ภาพที่ 4.6 แสดงผลการทดลองการวัดประสิทธิภาพการเข้าถึงไฟล์ของ NHAR โดยแบ่งการทดลองออกตามจำนวนไฟล์และจำนวน index file ซึ่งจะเห็นว่าหากจำนวนไฟล์ที่อยู่ภายใน NHAR มีจำนวน 20,000 – 40,000 ไฟล์ สามารถแบ่ง index ออกเป็น 2 ไฟล์ (แต่ละ index มีจำนวนไฟล์ไม่เกิน 20,000 ไฟล์) ได้โดยไม่กระทบกับประสิทธิภาพในการเข้าถึง แต่เมื่อไฟล์มีจำนวนมากกว่า 60,000 ไฟล์การแบ่ง index ออกเป็น 2 ไฟล์ (แต่ละ index มีจำนวนไฟล์มากกว่า 20,000 ไฟล์) จะทำให้ประสิทธิภาพในการเข้าถึงไฟล์ช้าลงอย่างเห็นได้ชัด ดังนั้นเพื่อให้ระบบสามารถทำงานได้อย่างมีประสิทธิภาพสำหรับสภาพแวดล้อมในงานวิจัยนี้ แต่ละ index block ของ NHAR ควรจะจัดเก็บไฟล์ไม่เกิน 20,000 ไฟล์ ดังนั้นเราสามารถกำหนดจำนวน index file เพื่อให้มีความเหมาะสมได้ดังนี้

$$Ni = ceil(\frac{Nf}{20,000}) \tag{4.1}$$

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

โดยที่; Nf คือจำนวนไฟล์ทั้งหมด

เนื่องจากในงานวิจัยนี้ ทางผู้วิจัยทำการทดลองจากไฟล์ที่อยู่ภายในระบบจำนวน 80,000 ไฟล์ ดังนั้น index file ในการทดลองจะถูกกำหนดให้มีค่าเป็น 4

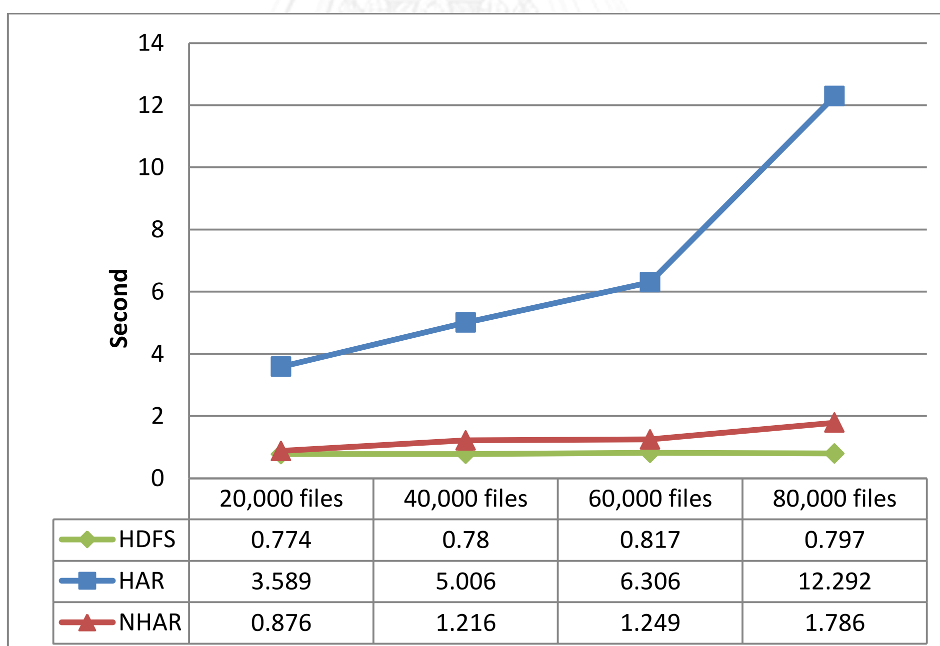
4.3.1 รูปแบบการทดลอง

ในการทดลองของงานวิจัยนี้จะมีรูปแบบที่ใช้สำหรับการทดลองดังนี้

- ทำการทดลองจากไฟล์สุ่ม (random file) จำนวน 10 ไฟล์ และทดลองซ้ำทั้งหมด 10 ครั้ง แล้วทำการหาค่าเฉลี่ยแต่แต่ละกลุ่มจำนวนข้อมูลที่ทำการทดลอง
- กลุ่มจำนวนข้อมูลที่ใช้ในการทำการทดลองจะมีจำนวน 20 000, 40 000, 60 000, 80 000 ไฟล์
- ใช้ค่า Ni = 4 เพราะในงานวิจัยนี้มีจำนวนไฟล์สูงสุด 80,000 ไฟล์ ซึ่งเมื่อคำนวนจากสมการที่ 4.1 แล้วจะมีค่าเป็น 4

4.3.2 การทดสอบประสิทธิภาพการเข้าถึงไฟล์ขนาดเล็ก

ในการทดลองประสิทธิภาพในการเข้าถึงไฟล์ขนาดเล็กนั้น ทางผู้วิจัยทำการทดลองและเปรียบเทียบจากการอ่านไฟล์จาก HDFS โดยตรง, การอ่านไฟล์จาก HAR และการอ่านไฟล์จาก NHAR โดยผลการทดลองถูกแสดงอยู่ในภาพที่ 4.7

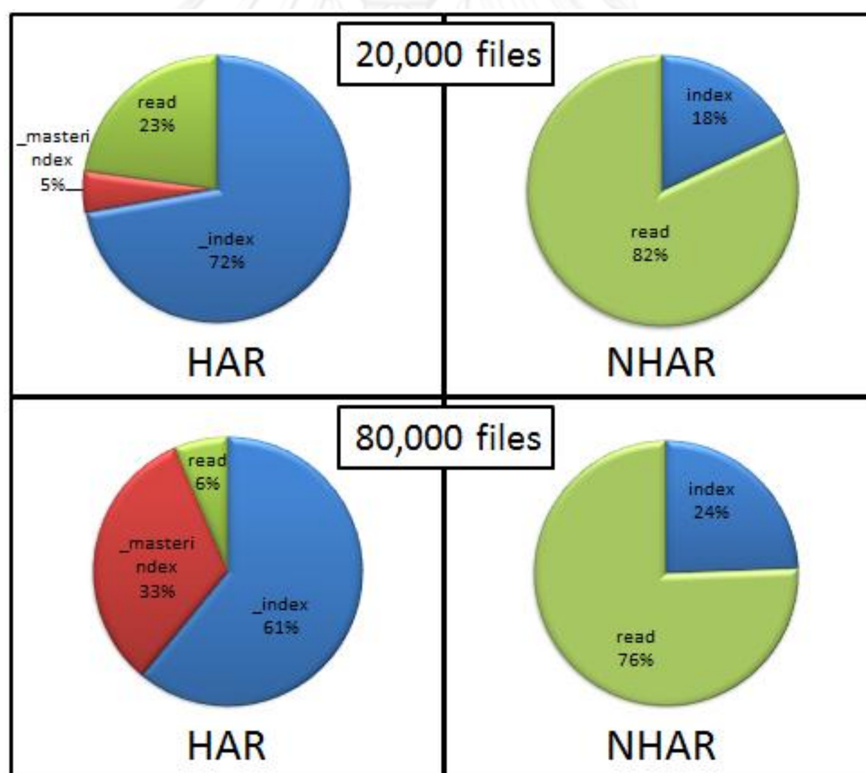


	20,000 files	40,000 files	60,000 files	80,000 files
HDFS	0.774	0.78	0.817	0.797
HAR	3.589	5.006	6.306	12.292
NHAR	0.876	1.216	1.249	1.786

ภาพที่ 4.7 กราฟผลการทดลองประสิทธิภาพในการเข้าถึงไฟล์ขนาดเล็ก

ภาพที่ 4.7 แสดงผลการทดลองประสิทธิภาพในการเข้าถึงไฟล์ขนาดเล็กโดยการเข้าถึงไฟล์จาก HDFS โดยตรง และการเข้าถึงไฟล์จาก archive file จำนวน 20 000, 40 000, 60 000 และ 80 000 ไฟล์ของ HAR และ NHAR จะเห็นว่าในการเข้าถึงไฟล์จาก HDFS โดยตรงจะมีประสิทธิภาพ

ที่ดีที่สุด รองลงมาคือ NHAR แต่ถึงแม้ว่าประสิทธิภาพในการเข้าถึงไฟล์จาก HDFS โดยตรงนั้นจะมีประสิทธิภาพมากกว่า NHAR แต่ในการจัดเก็บไฟล์ขนาดเล็กเอาไว้ใน HDFS จะมีผลกระทบในเรื่อง memory consumption อย่างมากดังแสดงอยู่ในหัวข้อที่ 4.5.5 ส่วนการเข้าถึงไฟล์จาก HAR จะมีประสิทธิภาพที่น้อยที่สุด ซึ่งเหตุผลที่ HAR นั้นมีประสิทธิภาพไม่ดีเท่าที่ควรนั้นเนื่องมาจาก HAR นั้นมีโครงสร้างเป็นแบบ two-level indexes และเนื่องจาก masterindex นั้นจะมีประสิทธิภาพในการเข้าถึงเป็น O(n) ซึ่งจากที่ได้กล่าวไปแล้วในหัวข้อที่ 3.1.3 นั่นคือยิ่งไฟล์มีจำนวนมากเท่าใด เวลาที่ใช้ในการค้นหาข้อมูลจาก masterindex file ก็จะยิ่งเพิ่มมากขึ้น ในขณะที่ NHAR นั้นจะเปลี่ยนจากการค้นหาจาก masterindex เป็นการทำ hashing ซึ่งคือการทำการ mod ตามจำนวน index file ซึ่งจะแบ่ง index file ออกเป็นหลาย index file และการเข้าถึงไฟล์ก็สามารถค้นหาจาก index file ที่จัดเก็บข้อมูลเอาไว้ได้โดยตรง ดังนั้นประสิทธิภาพในการเข้าถึงจึงเปลี่ยนจาก O(n) เป็น O(1) กล่าวคือหากเทียบระหว่างการไล่ค้นหาไฟล์ภายใน masterindex ตั้งแต่ต้นจนจบเพื่อดูว่าควรไปไล่ค้นหาที่ส่วนไหนของ index file แต่สำหรับ NHAR นั้นจะเป็นการทำ hashing ซึ่งคือการค้นหาเพียง 1 ครั้งแล้วสามารถไปค้นหาที่ index file ได้เลย โดยผลจากการวิเคราะห์เวลาที่ใช้ในการเข้าถึงในแต่ละส่วนระหว่าง HAR และ NHAR นั้นจะถูกแสดงอยู่ในภาพที่ 4.8



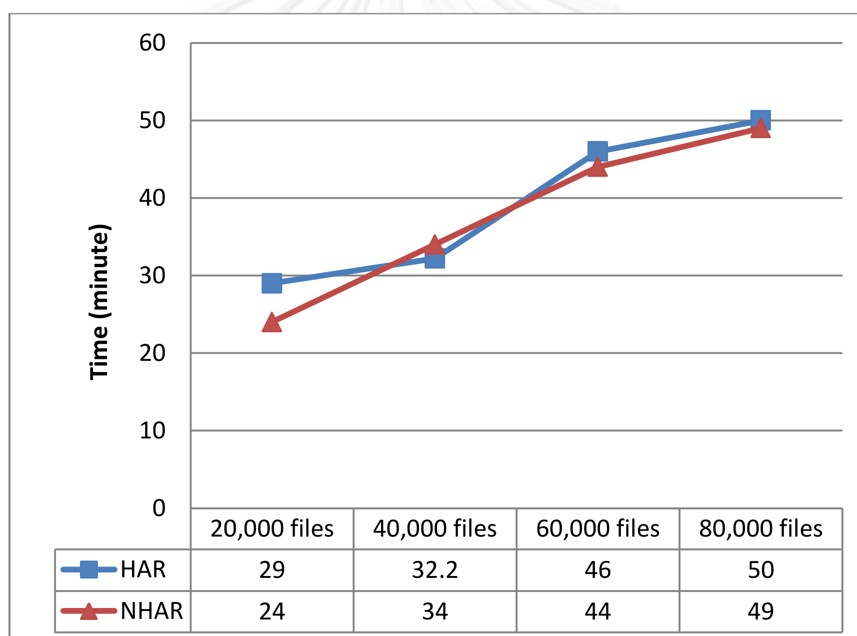
ภาพที่ 4.8 แสดงประสิทธิภาพในการเข้าถึงไฟล์ของ HAR และ NHAR

จากภาพที่ 4.8 เนื่องจากไฟล์ที่ใช้ทดสอบนั้นเป็นไฟล์เดียวกัน ดังนั้นเวลาที่ใช้ในการอ่านจะเท่ากัน แต่จะเห็นว่าการอ่านไฟล์ของ HAR นั้นจะเสียไปกับการอ่าน index file ทั้ง 2 index ซึ่งในการเข้าถึงไฟล์จำนวน 20,000 ไฟล์นั้นเวลาที่ HAR ใช้ส่วนใหญ่จะอยู่ที่ index file แต่เมื่อเข้าถึงไฟล์

จำนวน 80,000 ไฟล์ เวลาในการค้นหาภายใน masterindex file นั้นเริ่มใช้เวลามากขึ้น ในขณะที่เวลาในการค้นหาภายใน index file ของ NHAR นั้นใช้เวลาใกล้เคียงกัน โดยเมื่อทำเปรียบเทียบประสิทธิภาพในการเข้าถึงไฟล์ขนาดเล็กจำนวน 20 000, 40 000, 60 000 และ 80 000 ไฟล์ NHAR จะมีประสิทธิภาพมากกว่า HAR ถึง 75.59%, 75.71%, 80.19% และ 85.47% ตามลำดับ

4.3.3 การทดสอบประสิทธิภาพในการสร้าง NHAR

จากที่ได้กล่าวไปแล้วในบทที่ 3 เนื่องจากในงานวิจัยนี้นำหลักการในการทำงานของ HAR มาใช้เป็นพื้นฐานในการวิจัย โดยในการสร้าง HAR และ NHAR นั้นจะเป็นการเรียกใช้ MapReduce algorithm เช่นเดียวกัน ดังนั้นทางผู้วิจัยจึงทำการประเมินเวลาที่ใช้ในการสร้าง archive file ของ NHAR โดยทำการเปรียบเทียบเวลาที่ใช้กับ HAR แบบดั้งเดิม

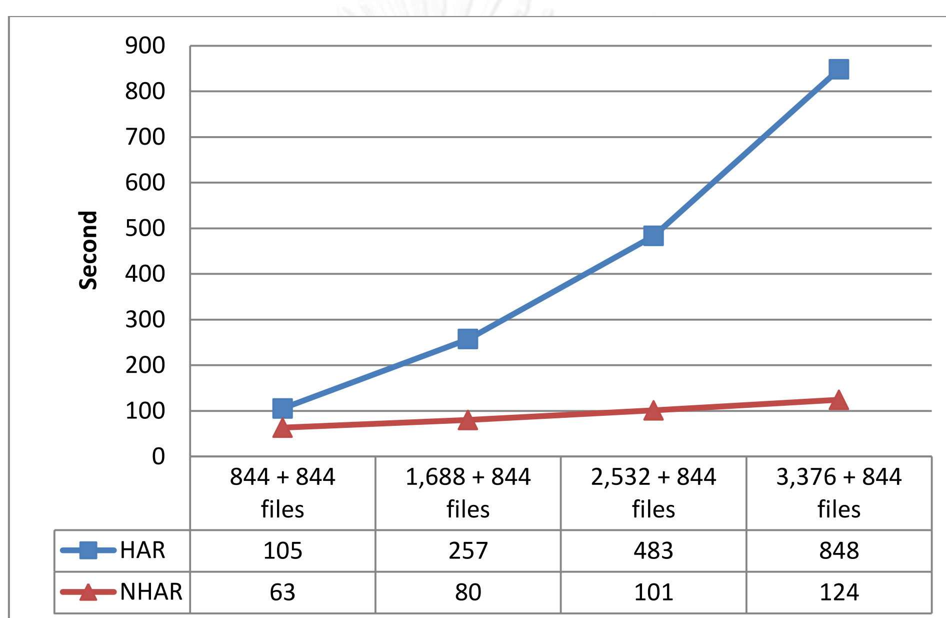


ภาพที่ 4.9 กราฟผลการทดลองการเปรียบเทียบเวลาในการสร้างไฟล์

ภาพที่ 4.9 แสดงเวลาที่ใช้ในการสร้างไฟล์ HAR และ NHAR โดยทำการทดลองการทำ archive ไฟล์ขนาดเล็กจำนวน 20 000, 40 000, 60 000 และ 80 000 ไฟล์ ตามลำดับ โดยเมื่อทำการเปรียบเทียบเวลาที่ใช้ในการสร้างไฟล์ของ HAR และ NHAR จะเห็นว่าเวลาที่ใช้ในการสร้าง archive file นั้นใช้เวลาใกล้เคียงกัน

4.3.4 การทดสอบประสิทธิภาพในการเพิ่มไฟล์ของ NHAR

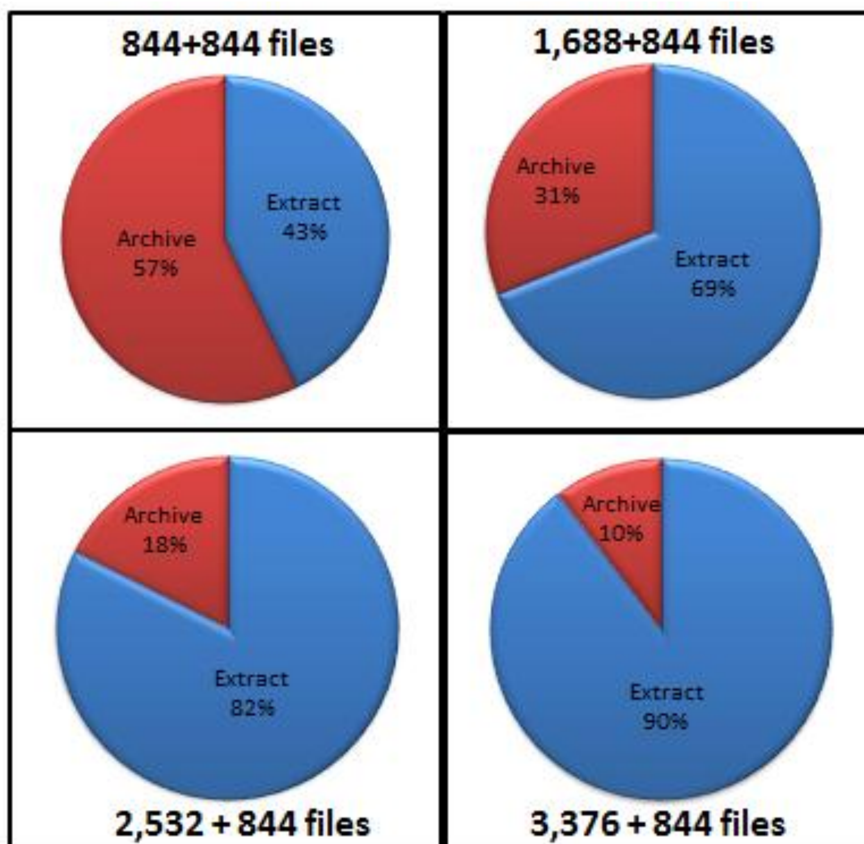
จากที่ได้กล่าวไปแล้วในหัวข้อที่ 3.2 ในการเพิ่มไฟล์เข้าไปยังไฟล์ HAR ที่มีอยู่แล้วนั้นผู้ใช้จะต้องทำการแตกไฟล์ (extract) ภายในออกมาทั้งหมดก่อน แล้วจึงค่อยทำการสร้าง HAR file ใหม่ขึ้นมาอีกครั้ง โดยรวมไฟล์ที่แตกออกมานั้นกับไฟล์ใหม่ที่ต้องการเพิ่มเข้าไป ในขณะที่ NHAR นั้น ผู้ใช้สามารถเรียกใช้งาน Inserting Function ได้ทันที ซึ่งจะช่วยลดเวลาที่ใช้และทำให้ระบบมีประสิทธิภาพมากยิ่งขึ้น



	844 + 844 files	1,688 + 844 files	2,532 + 844 files	3,376 + 844 files
HAR	105	257	483	848
NHAR	63	80	101	124

ภาพที่ 4.10 กราฟแสดงประสิทธิภาพในการเพิ่มไฟล์เข้าไปยังไฟล์ archive ที่มีอยู่แล้ว

ภาพที่ 4.10 แสดงการเปรียบเทียบประสิทธิภาพในการเพิ่มไฟล์ใหม่เข้าไปยัง archive file ที่มีอยู่แล้ว โดยในการทดลองจะเริ่มต้นจากการสร้างไฟล์ HAR และ NHAR ที่มีไฟล์จำนวน 844 ไฟล์ (ขนาดรวม 64MB) ก่อนเป็นไฟล์ตั้งต้น จากนั้นทำการเพิ่มไฟล์จำนวน 844 ไฟล์เข้าไปยังไฟล์ที่ถูกสร้างขึ้นก่อนหน้า ซึ่งเมื่อเพิ่มไฟล์แล้วจะทำไฟล์ archive มีจำนวนไฟล์รวมเป็น 1,688 ไฟล์ แล้วจึงทำการเพิ่มไฟล์เข้าไปจำนวน 844 ไฟล์อีกครั้ง และทำเช่นนี้ต่อไปเรื่อยๆ ซึ่งผลลัพธ์ที่ได้แสดงให้เห็นว่า Inserting Function ของ NHAR นั้นมีประสิทธิภาพดีกว่า HAR อย่างเห็นได้ชัด เนื่องจาก NHAR นั้นไม่ต้องเสียเวลาในส่วนของการแตกไฟล์ก่อน สามารถใส่ไฟล์ใหม่เพิ่มเข้าไปได้ทันที ในขณะที่ HAR นั้นต้องเสียเวลาในการแตกไฟล์ก่อน



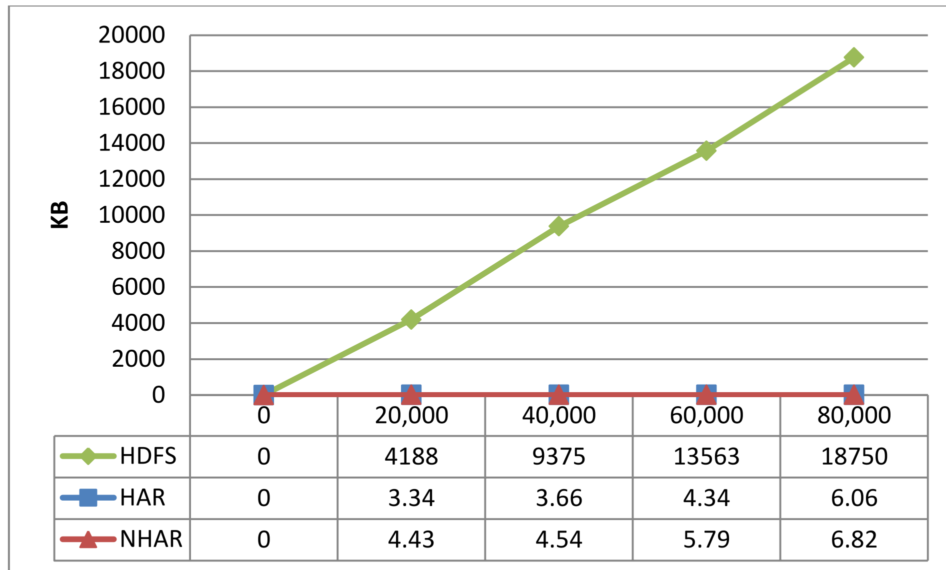
ภาพที่ 4.11 แสดงประสิทธิภาพในการเพิ่มไฟล์ของ HAR แบบดั้งเดิม

จากภาพที่ 4.11 แสดงประสิทธิภาพในการเพิ่มไฟล์ของ HAR แบบดั้งเดิม จากที่ได้กล่าวไปแล้วในหัวข้อที่ 3.2.2 เนื่องจาก HAR ต้องเสียเวลาในการแตกไฟล์ทั้งหมดออกมาก่อน จึงทำให้ประสิทธิภาพในการทำงานนั้นไม่ดีเท่าที่ควร ซึ่งจะเห็นว่าหากเป็นการแตกไฟล์ที่มีจำนวนน้อยเวลาที่ใช้ในการทำ archive และแตกไฟล์จะไม่ต่างกัน แต่เมื่อไฟล์มีจำนวนที่มากขึ้นจะเห็นได้ว่าการแตกไฟล์นั้นถือเป็นปัญหาอย่างเห็นได้ชัด

ดังนั้นจากผลการทดลองเมื่อทำการเปรียบเทียบ HAR และการใช้งาน Inserting Function ของ NHAR นั้น NHAR สามารถลดเวลาที่ใช้ได้มากถึง 40%, 68.87%, 79.09% และ 85% ตามลำดับ

4.3.5 การทดสอบการใช้งานพื้นที่เก็บ (Memory usage) บน NameNode

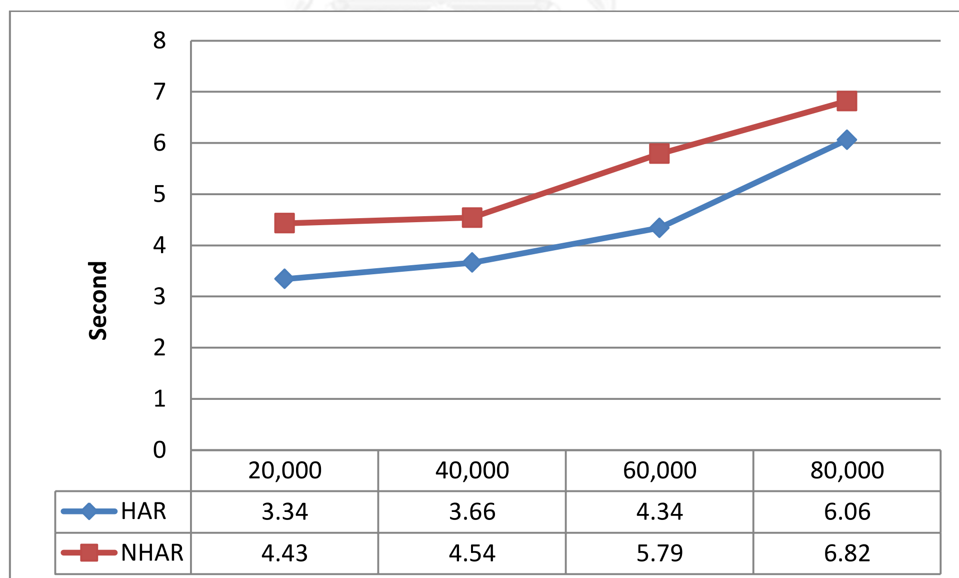
ในการประเมินประสิทธิภาพในการใช้พื้นที่บน NameNode ในการจัดเก็บไฟล์ขนาดเล็กนั้น โดยกลุ่มของข้อมูลที่ใช้ในการทดลองจะประกอบด้วยไฟล์จำนวน 20 000 ไฟล์, 40 000 ไฟล์, 60 000 ไฟล์ และ 80 000 ไฟล์ โดยแต่ละกลุ่มของข้อมูลจะถูกจัดเก็บอยู่ภายใน directory 10 directories, 20 directories, 30 directories และ 40 directories ตามลำดับ



	0	20,000	40,000	60,000	80,000
HDFS	0	4188	9375	13563	18750
HAR	0	3.34	3.66	4.34	6.06
NHAR	0	4.43	4.54	5.79	6.82

ภาพที่ 4.12 กราฟแสดงการใช้งานพื้นที่บน NameNode

ทางผู้วิจัยทดการทดลองเพื่อทดสอบพื้นที่ที่ใช้ในการจัดเก็บ metadata บน NameNode โดยผลลัพธ์ถูกแสดงอยู่ในภาพที่ 4.12 ซึ่งผลลัพธ์ที่ได้แสดงให้เป็นว่า HAR และ NHAR นั้นสามารถจัดเก็บไฟล์ขนาดเล็กได้อย่างมีประสิทธิภาพมากกว่าการจัดเก็บบน HDFS แบบดั้งเดิม



	20,000	40,000	60,000	80,000
HAR	3.34	3.66	4.34	6.06
NHAR	4.43	4.54	5.79	6.82

ภาพที่ 4.13 แสดงการเปรียบเทียบการใช้งานพื้นที่บน NameNode ระหว่าง HAR และ NHAR

ภาพที่ 4.13 แสดงกราฟขยายจากภาพที่ 4.12 โดยแสดงการเปรียบเทียบการใช้งานพื้นที่บน NameNode ของ HAR และ NHAR โดยเมื่อทำการเปรียบเทียบระหว่าง HAR และ NHAR จะเห็นว่า NHAR ใช้พื้นที่ในการจัดเก็บมากกว่า HAR เล็กน้อย เนื่องจาก NHAR นั้นมีการออกแบบ index ให้ใช้งาน index 4 ไฟล์ ในขณะที่ HAR นั้นใช้ index เพียงแค่ 2 ไฟล์ ซึ่งนั่นทำให้ NHAR เกิด overhead บน NameNode เพิ่มมากขึ้น

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

บทที่ 5
บทสรุปและแนวทางในการพัฒนาต่อ

5.1 บทสรุป

เนื่องจาก HDFS นั้นถูกออกแบบมาสำหรับจัดการกับไฟล์ขนาดใหญ่ จึงทำให้ไม่เหมาะสำหรับการจัดการไฟล์ขนาดเล็ก ซึ่งเหตุผลที่ทำให้ HDFS ไม่เหมาะสำหรับไฟล์ขนาดเล็กนั้นเนื่องมาจากโครงสร้างของ HDFS นั้นเป็นแบบ single master-multipls slaves โดยข้อจำกัดที่สำคัญที่ทำให้ HDFS นั้นไม่สามารถขยายระบบได้นั้นคือหน่วยความจำ (Memory) บน NameNode มีขีดจำกัดตามหน่วยความจำทางกายภาพ (Physical memory) ของ NameNode เนื่องจากมี master เพียงหนึ่งตัว เมื่อมีการจัดเก็บไฟล์ขนาดเล็กจำนวนมากเอาไว้บน HDFS ทำให้ต้องใช้หน่วยความจำขนาดใหญ่เพื่อจัดเก็บ metadata ของไฟล์ขนาดเล็กจำนวนมากเหล่านั้น ซึ่งบล็อกแต่ละบล็อกนั้นจะจัดเก็บไฟล์เพียง 1 ไฟล์ จึงทำให้เกิดบล็อกขนาดเล็กจำนวนมาก (ขนาดเล็กกว่าขนาดบล็อกที่กำหนด) ดังนั้น NameNode จึงต้องคอยรับการร้องขอที่อยู่ในการจัดเก็บจำนวนมาก และการร้องของการกระจาย storage block บ่อยขึ้น ซึ่งทำให้ประสิทธิภาพของการรองรับ I/O ขนาดเล็กเหล่านั้นเกิดปัญหาคอขวด (Bottleneck) ขึ้น นอกจากนั้นแล้วยังทำให้ประสิทธิภาพแย่ลงเมื่อต้องจัดการกับไฟล์ขนาดเล็กจำนวนมากอีกด้วย

โดยในงานวิจัยนี้ ทางผู้วิจัยนำเสนอกลไกที่เรียกว่า New Hadoop Archive หรือ NHAR ที่ซึ่งทำการออกแบบโครงสร้างของ HAR ขึ้นมาใหม่เพื่อเพิ่มประสิทธิภาพในการเข้าถึงไฟล์ขนาดเล็กบน Hadoop เนื่องมาจากการอ่านไฟล์จาก HAR นั้นจะต้องทำการเข้าถึง index ไฟล์ 2 ครั้งซึ่งทำให้มีผลกระทบกับประสิทธิภาพในการเข้าถึง ดังนั้นทางผู้วิจัยจึงทำการแก้ไขโครงสร้างของ HAR เพื่อลดขั้นตอนในการจัดเก็บ metadata ของไฟล์ขนาดเล็กและเพื่อเพิ่มประสิทธิภาพในการเข้าถึงให้ดีมากยิ่งขึ้น โดยทำการออกแบบโครงสร้าง index file ขึ้นมาใหม่ โดยเปลี่ยนการใช้งาน master-index เป็นการสร้าง hash table จากข้อมูล index และแบ่งข้อมูลเหล่านั้นให้มีหลาย index ไฟล์แทน ซึ่งผลการทดลองแสดงให้เห็นว่า NHAR สามารถเพิ่มประสิทธิภาพของ I/O ขนาดเล็กได้เป็นอย่างดี โดยเมื่อทำการเปรียบเทียบประสิทธิภาพในการเข้าถึงกับ HAR แบบดั้งเดิมนั้น NHAR สามารถเข้าถึงได้ดีกว่ามากถึง 85.47% นอกจากนั้นแล้วในงานวิจัยนี้ยังขยายขีดจำกัดความสามารถของ HAR ที่ซึ่งไม่สามารถแก้ไขไฟล์ที่อยู่ภายใน HAR ที่ถูกสร้างไปแล้วได้ โดยในงานวิจัยนี้ ทางผู้วิจัยทำการเพิ่มการทำงานของ NHAR ให้สามารถเพิ่มไฟล์เข้าไปยัง NHAR ที่ถูกสร้างไปแล้วได้ แต่การเพิ่มไฟล์เข้าไปจะทำให้มี part file ใหม่เพิ่มเข้ามาเสมอ ดังนั้นในเพิ่มไฟล์ โปรแกรมจะมีการแจ้งเตือนเพื่อให้ผู้ใช้งานทราบว่าหลังจากทำการเพิ่มไฟล์แล้วจะทำให้มี part file รวมอยู่ทั้งหมดกี่ part file และหากทำการ reorganize จะสามารถลดจำนวน part file ไปได้เหลือกี่ไฟล์ โดยผลจากการทดลองแสดงให้เห็นว่าการใช้งาน inserting function สามารถลดเวลาที่ใช้ในการ archive ได้มากถึง 85%

5.2 ข้อจำกัด

ข้อจำกัดในงานวิจัยนี้มีดังต่อไปนี้

1. เนื่องจากเครื่อง cluster ที่ใช้ในการทดลองนั้นสร้างขึ้นมาจากเครื่อง 5 เครื่อง ซึ่งถือว่าเป็นระบบขนาดเล็ก ดังนั้นเมื่อทำการทดลองกับไฟล์จำนวนมากกว่า 100,000 ไฟล์ขึ้นไปเครื่องจะทำงานหนัก และส่งผลให้ประสิทธิภาพในการทดลองแย่ลงอย่างมาก
2. เครื่องมือนี้สามารถใช้งานได้กับไฟล์ที่ถูกจัดเก็บเอาไว้ภายใน HDFS อยู่แล้ว ไม่สามารถใช้งานกับไฟล์ที่อยู่ใน local ได้ ดังนั้นหากต้องการใช้งานเครื่องมือนี้จะต้องทำการย้ายไฟล์ที่อยู่ใน local ไปไว้ภายใน HDFS ก่อน
3. ในระหว่างที่ทำการ archive ไฟล์จำนวนมากกว่า 30,000 ไฟล์ขึ้นไป หากมีผู้ใช้งานระบบอยู่เป็นจำนวนมาก จะทำให้ประสิทธิภาพในการทำงานแย่ลง
4. เมื่อ archive file ถูกสร้างขึ้นแล้วจะไม่สามารถลบไฟล์ที่อยู่ภายในได้

5.3 แนวทางในการพัฒนาต่อ

เครื่องมือที่จัดทำขึ้นนี้ยังสามารถทำการพัฒนาต่อไปได้อีก เพื่อให้มีประสิทธิภาพในการทำงานมากยิ่งขึ้น ดังนี้

1. สร้างสภาพแวดล้อมในการทดลองให้มีขนาดใหญ่ขึ้น และทำการพัฒนาเครื่องมือให้สามารถใช้งานกับไฟล์ที่มีจำนวนมากขึ้นได้
2. พัฒนาให้สามารถใช้งานกับไฟล์ที่อยู่ภายนอก HDFS ได้
3. พัฒนาโครงสร้างการทำ index ของไฟล์ขนาดเล็กให้ดีมากยิ่งขึ้น เพื่อให้ประสิทธิภาพในการเข้าถึงไฟล์เทียบเท่ากับการเข้าถึงไฟล์จาก HDFS โดยตรง หรือดีมากยิ่งขึ้น
4. เพิ่ม function การจัดการไฟล์ภายใน archive file ให้มีความสามารถเสมือน file system ทั่วไป หรือเทียบเท่าการทำงานของ HDFS ดั้งเดิม เช่นเพิ่มความสามารถในการลบไฟล์ที่อยู่ภายใน archive file ได้, สามารถคัดลอกหรือย้ายไฟล์จาก local เข้าไปยัง archive file ที่ถูกสร้างขึ้นมาแล้วได้, สร้างสารบบและจัดการข้อมูลที่อยู่ภายใน archive file ได้ เป็นต้น
5. ทำการพัฒนาระบบให้สามารถรวมไฟล์ได้แบบอัตโนมัติ โดยอาจมีการกำหนดว่าเมื่อไฟล์มีจำนวนเกินกว่าจำนวนที่กำหนดแล้วให้ทำการสร้าง NHAR file อัตโนมัติ
6. พัฒนาระบบให้มีการเข้ารหัส โดยหากไฟล์มีความสำคัญ หรือเป็นข้อมูลที่เป็นความลับอาจสามารถกำหนดให้ NHAR file เข้ารหัส เพื่อเพิ่มความปลอดภัยให้กับไฟล์ที่อยู่ภายใน NHAR file

รายการอ้างอิง

1. Vrable, M., S. Savage, and G.M. Voelker, *Cumulus: File system back up to the cloud.* ACM Transactions on Storage (TOS), 2009. 5(4).
2. Ghemawat, S., H. Gobioff, and S.-T. Leung, *The Google file system.* SIGOPS Oper. Syst. Rev., 2003. 37(5): p. 29-43.
3. Shafer, J., S. Rixner, and A.L. Cox. *The Hadoop distributed filesystem: Balancing portability and performance*. in *Performance Analysis of Systems & Software (ISPASS),* 2010 *IEEE International Symposium on*. 2010.
4. Amazon-S3, *Amazon simple storage service* (*amazon s*3). http://www.amazon.com/s, 2009.
5. *Apache Hadoop for Archiving Email.* http://blog.cloudera.com/blog/2011/09/hadoop-for-archiving-email/.
6. *http://en.wikipedia.org/wiki/Apache_Hadoop.*
7. *HDFS Federation.* http://hadoop.apache.org/docs/stable2/hadoop-project-dist/hadoop-hdfs/Federation.html.
8. *An Introduction to HDFS Federation.* http://hortonworks.com/blog/an-introduction-to-hdfs-federation/.
9. Liu, J., L. Bing, and S. Meina. *THE optimization of HDFS based on small files*. in *Broadband Network and Multimedia Technology (IC-BNMT),* 2010 *3rd IEEE International Conference on*. 2010.
10. Dong, B., et al. *A Novel Approach to Improving the Efficiency of Storing and Accessing Small Files on Hadoop: A Case Study by PowerPoint Files*. in *Services Computing* (*SCC*), 2010 *IEEE International Conference on*. 2010.
11. Dong, B., et al., *An optimized approach for storing and accessing small files on cloud storage.* Journal of Network and Computer Applications, 2012. 35(6): p. 1847-1862.
12. Xuhui, L., et al. *Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS*. in *Cluster Computing and Workshops,* 2009. *CLUSTER '*09. *IEEE International Conference on*. 2009.
13. *Hadoop Archive.* http://hadoop.apache.org/docs/r1.2.1/hadoop_archives.html.
14. Mackey, G., S. Sehrish, and W. Jun. *Improving metadata management for small files in HDFS*. in *Cluster Computing and Workshops,* 2009. *CLUSTER '*09. *IEEE International Conference on*. 2009.

15. Borthakur, D., *The Hadoop Distributed File System: Architecture and Design.* Hadoop Documentation, 2007.
16. HEDLUND, B., *Understanding Hadoop Clusters and the Network.* http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/.
17. Sze, T.-W.N. and M. Konar, *The Problem of Many Small Files.* http://developer.yahoo.com/blogs/hadoop/hadoop-archive-file-compaction-hdfs-461.html.
18. White, T., *The Small Files Problem.* http://blog.cloudera.com/blog/2009/02/the-small-files-problem/.
19. Dean, J. and S. Ghemawat, *MapReduce: simplified data processing on large clusters.* Commun. ACM, 2008. 51(1): p. 107-113.
20. *Big O notation.* http://th.wikipedia.org/wiki/%E0%B8%AA%E0%B8%B1%E0%B8%8D%E0%B8%81%E0%B8%A3%E0%B8%93%E0%B9%8C%E0%B9%82%E0%B8%AD%E0%B9%83%E0%B8%AB%E0%B8%8D%E0%B9%88.
21. *Algorithm Efficiency Analysis.* http://www.vcharkarn.com/lesson/view.php?id=46.
22. Shvachko, K., *Name-node memory size estimates and optimization proposal.* https://issues.apache.org/jira/browse/HADOOP-1687, 2007.

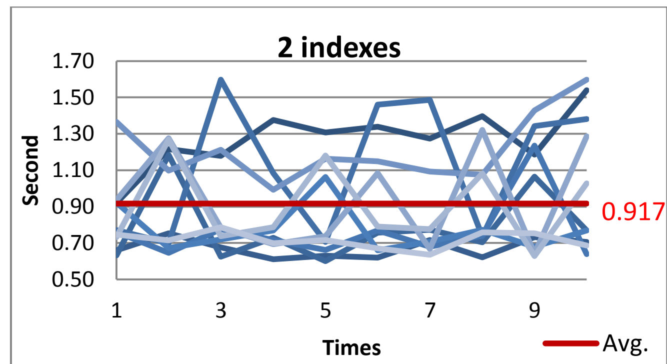
จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

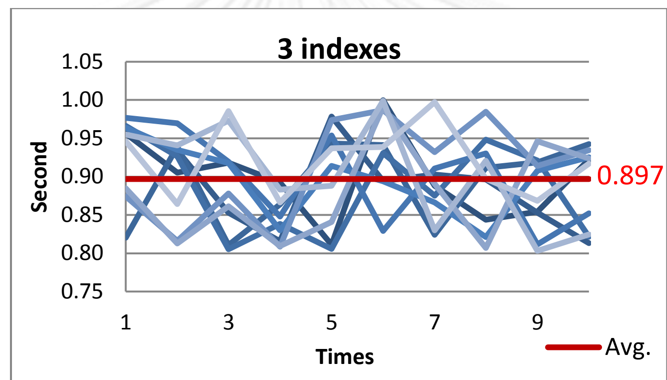
ภาคผนวก

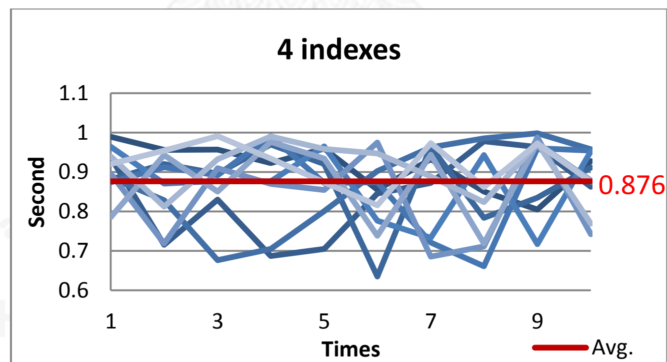
จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

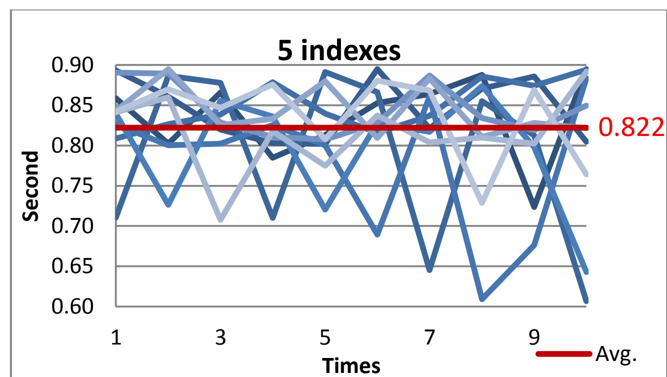
กราฟแสดงผลการทดลองการวัดประสิทธิภาพตามจำนวน index file

ไฟล์จำนวน 20K

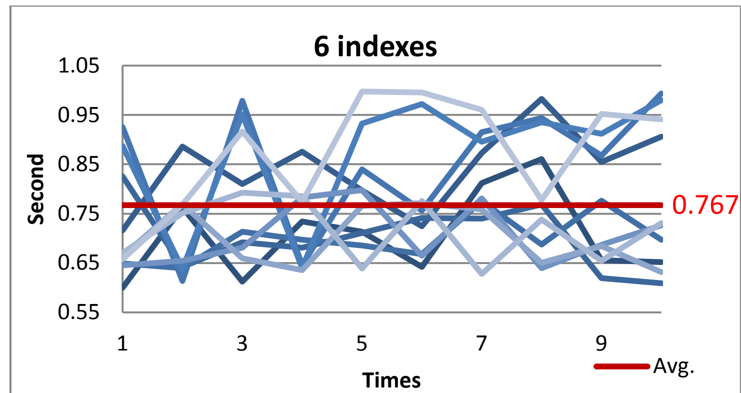




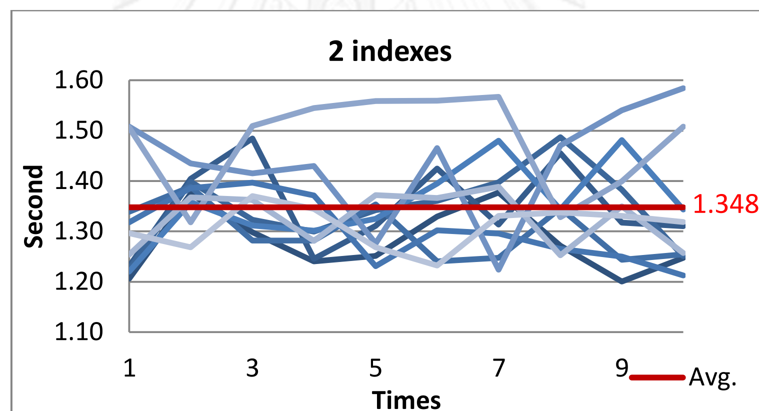


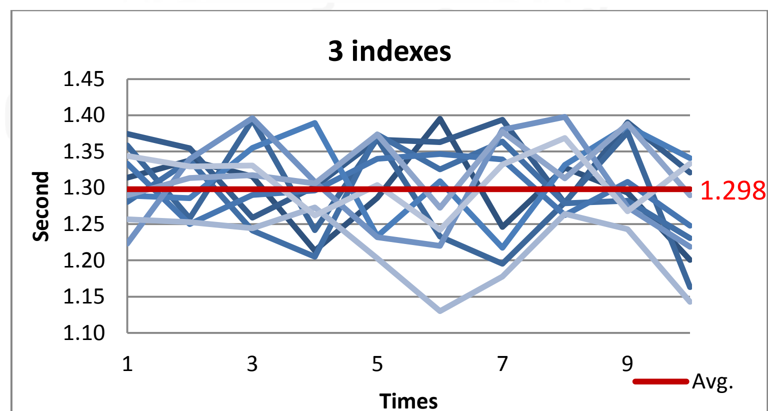


ไฟล์จำนวน 20K (ต่อ)

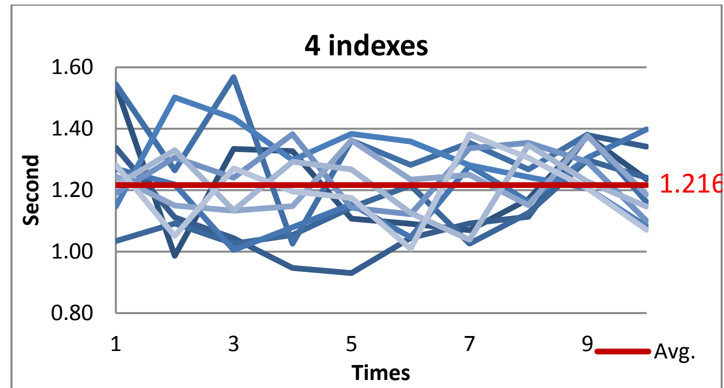


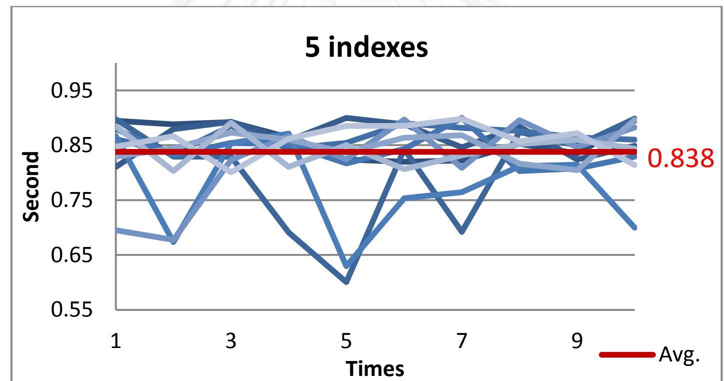
ไฟล์จำนวน 40K

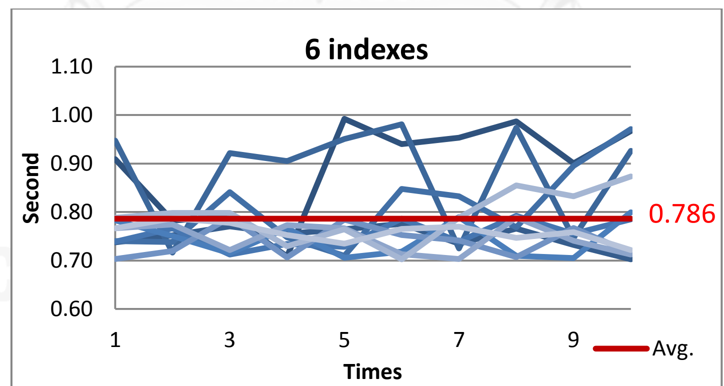




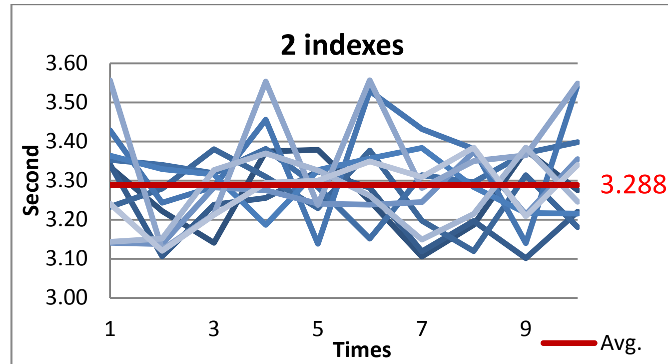
ไฟล์จำนวน 40K (ต่อ)

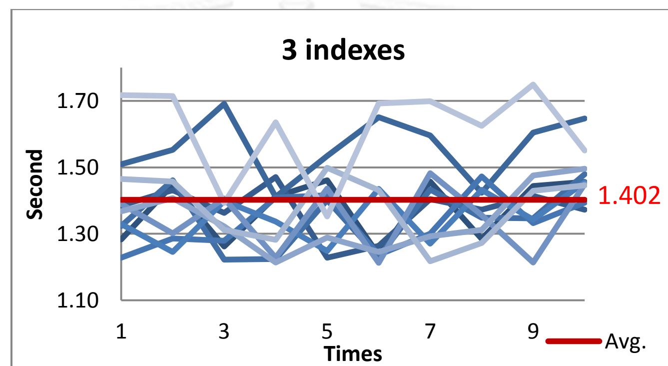


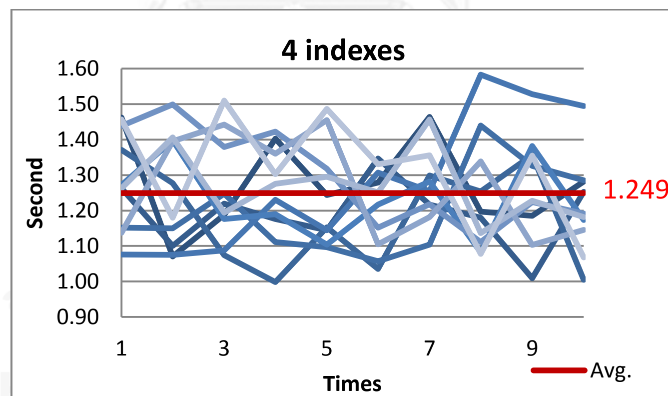


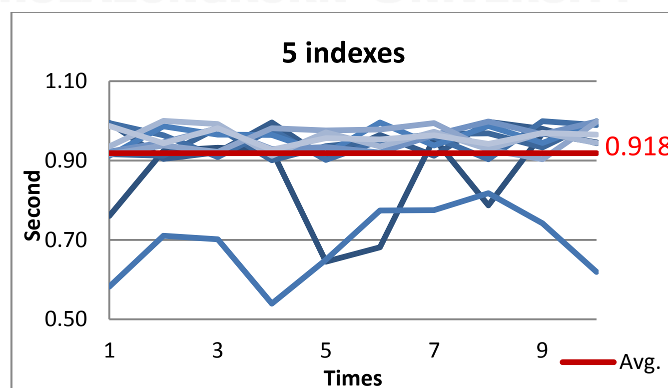


ไฟล์จำนวน 60K

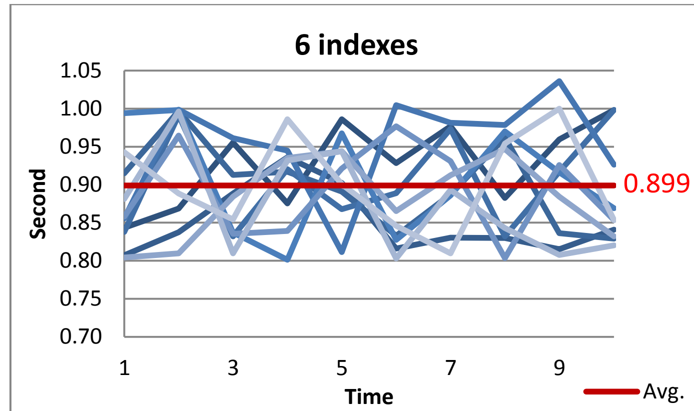




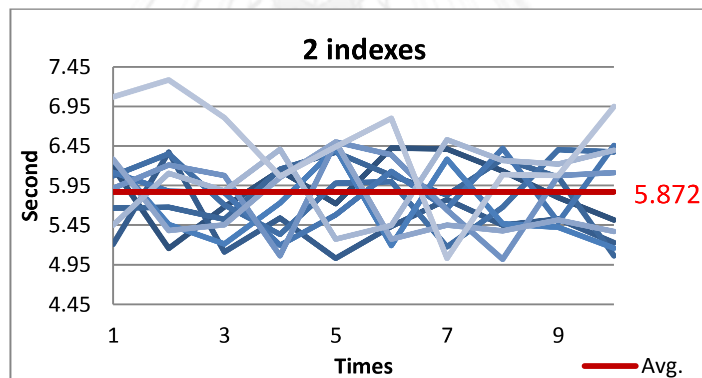


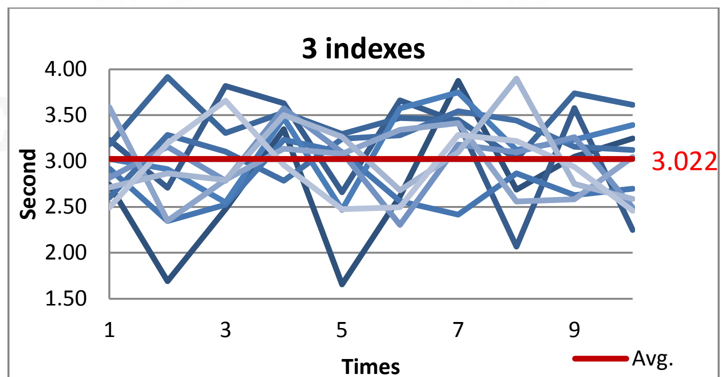


ไฟล์จำนวน 60K (ต่อ)

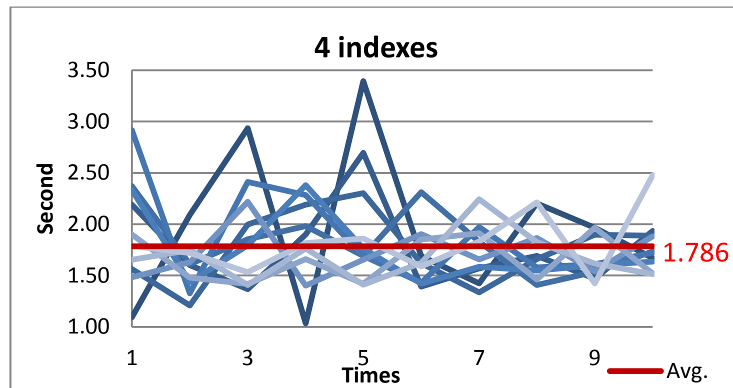


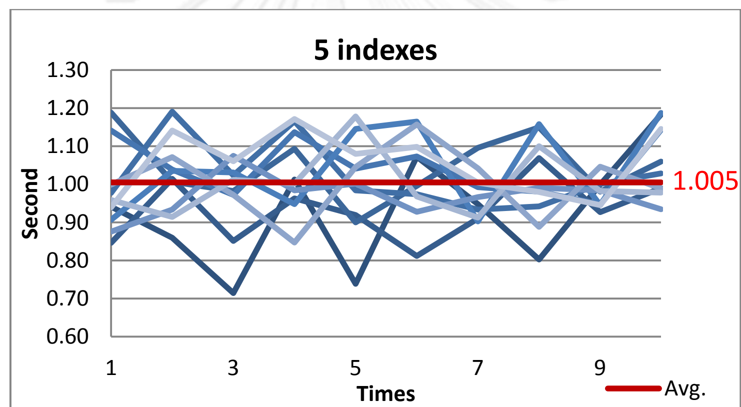
ไฟล์จำนวน 80K

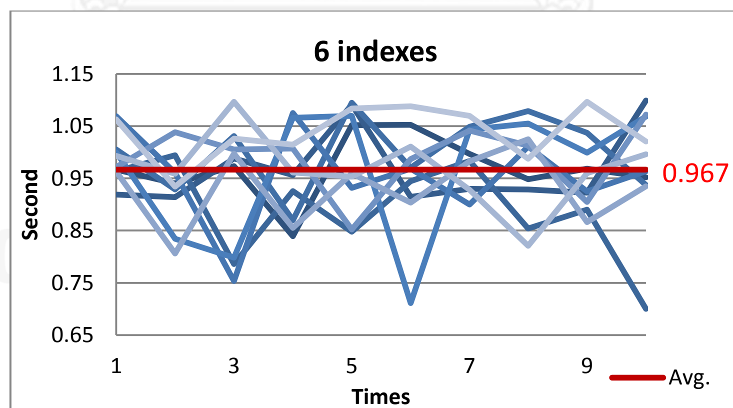




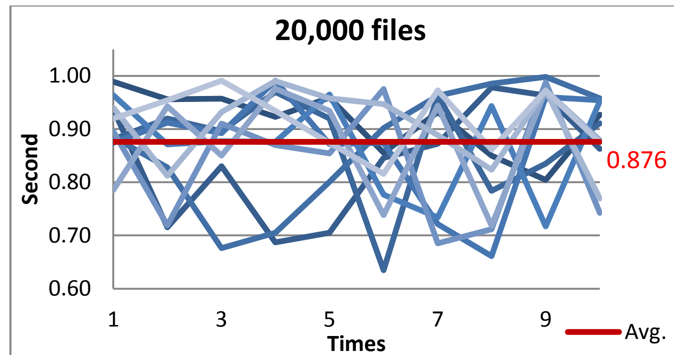
ไฟล์จำนวน 80K (ต่อ)

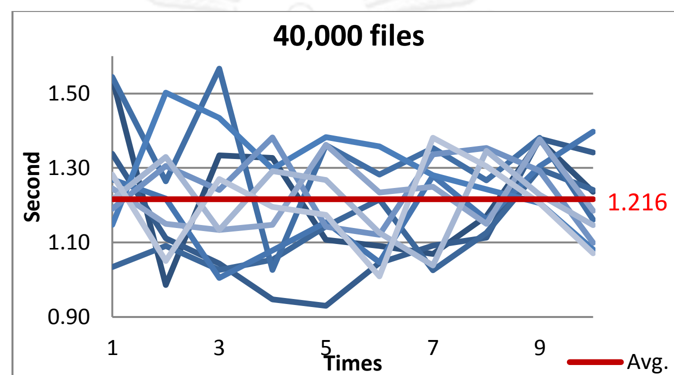


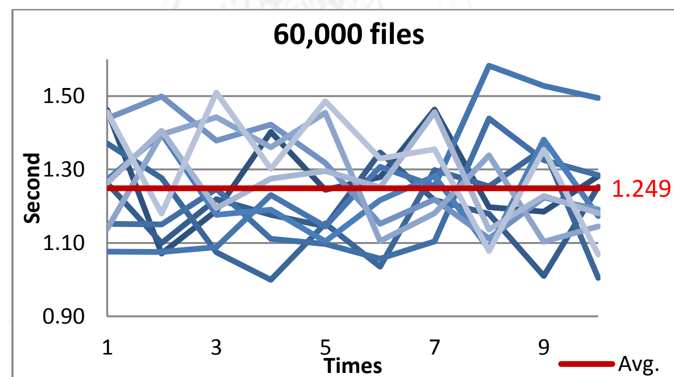


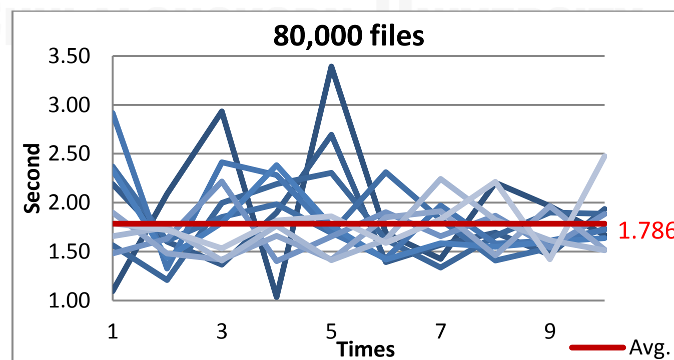


กราฟแสดงผลการทดลองประสิทธิภาพการเข้าถึงไฟล์ขนาดเล็ก

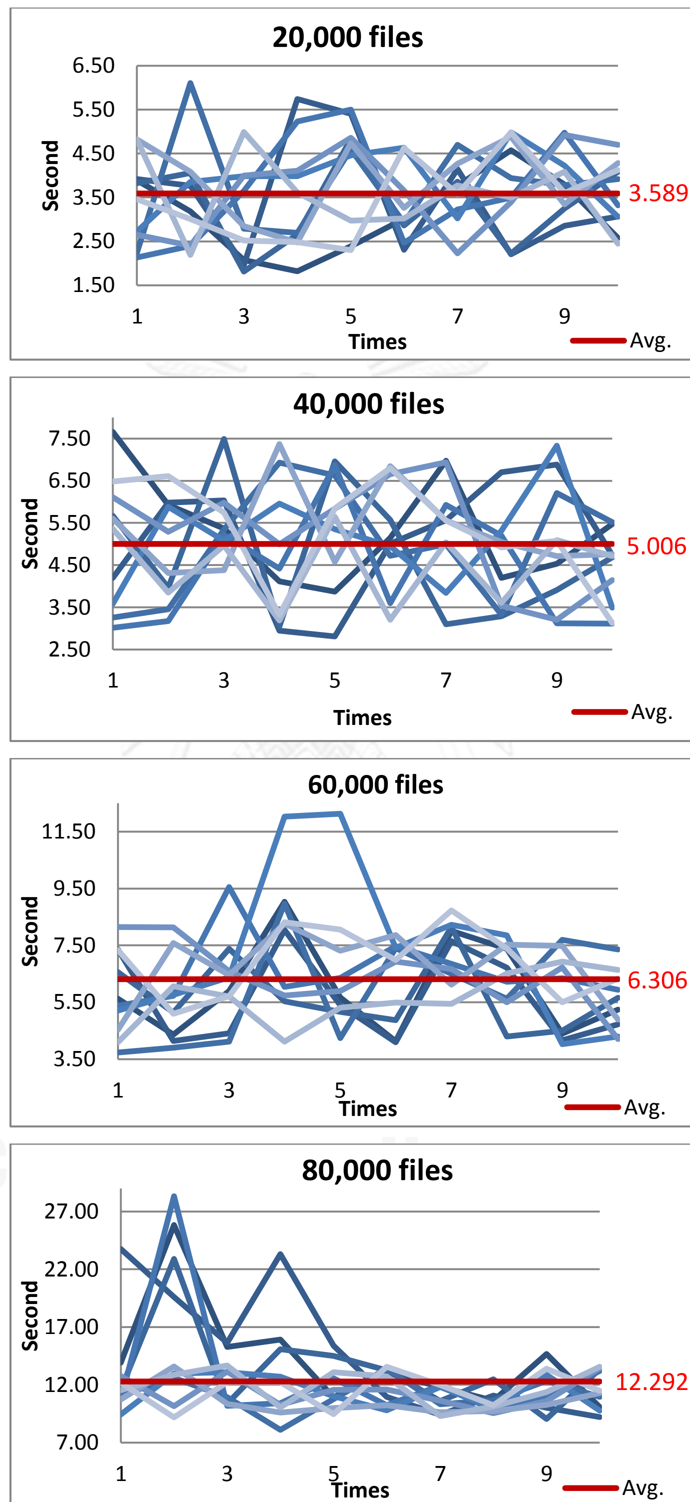




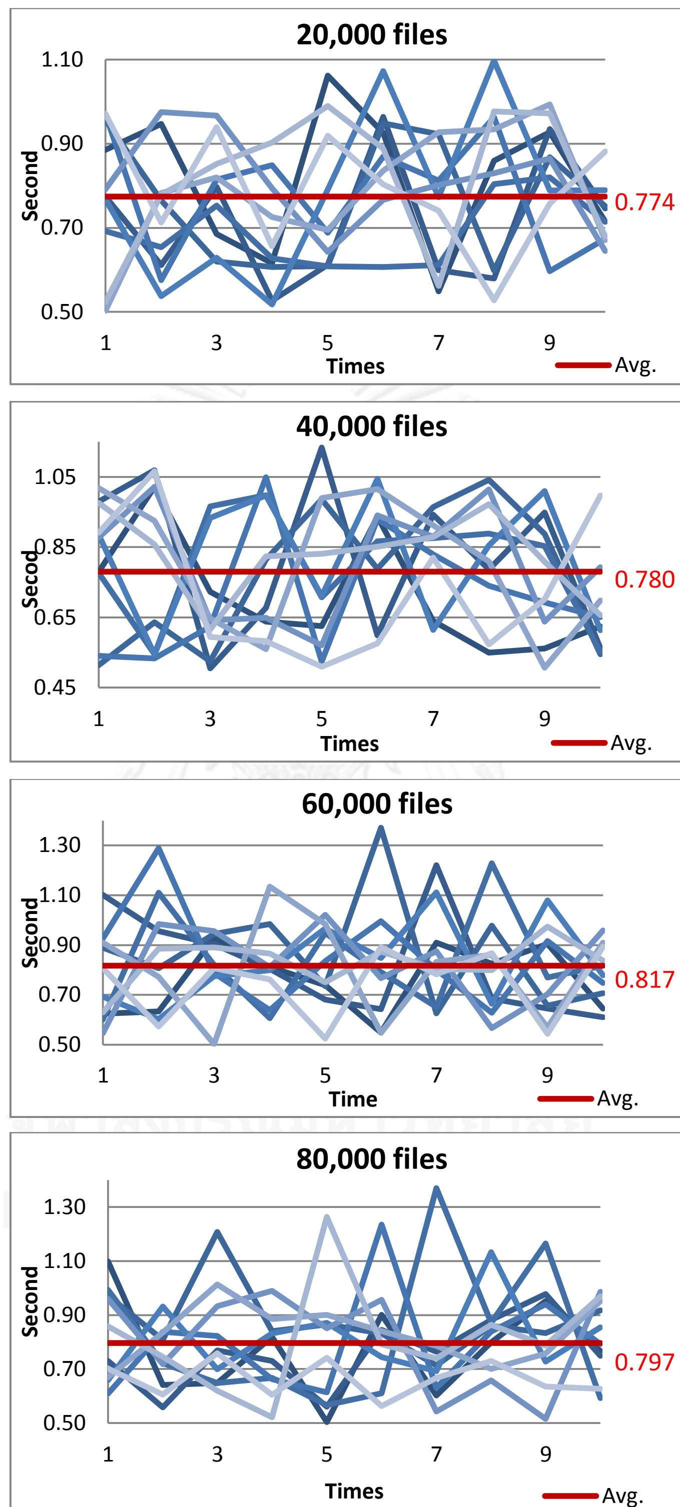




กราฟแสดงประสิทธิภาพการเข้าถึง NHAR file

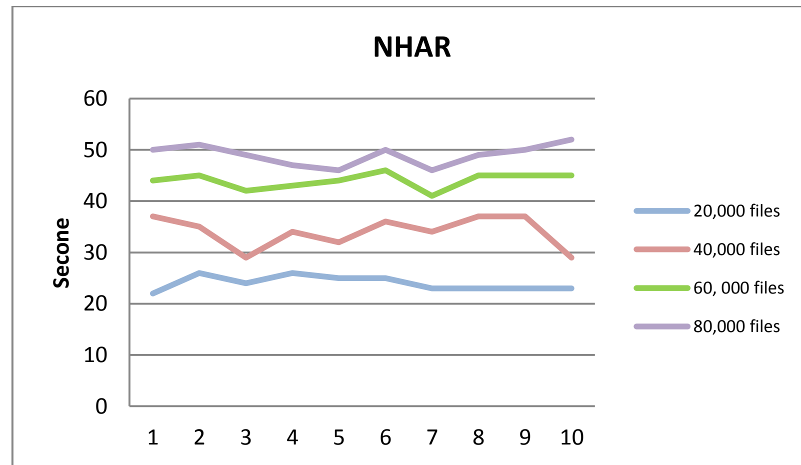


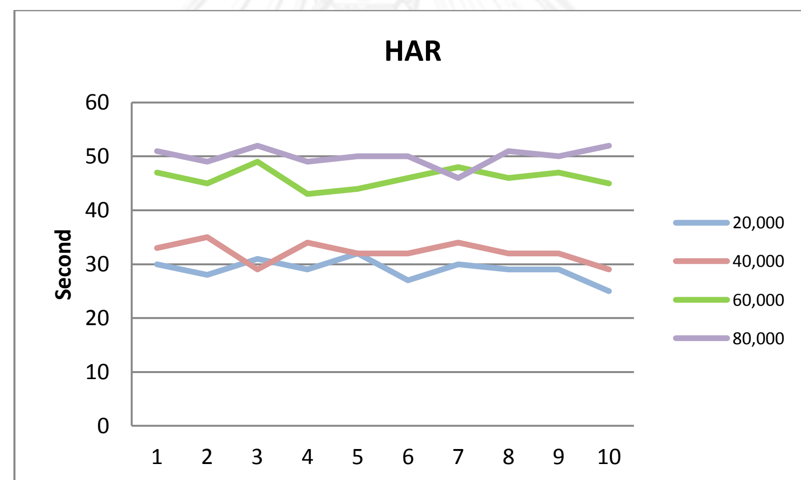
กราฟแสดงประสิทธิภาพการเข้าถึง HAR file



กราฟแสดงประสิทธิภาพการเข้าถึง HDFS

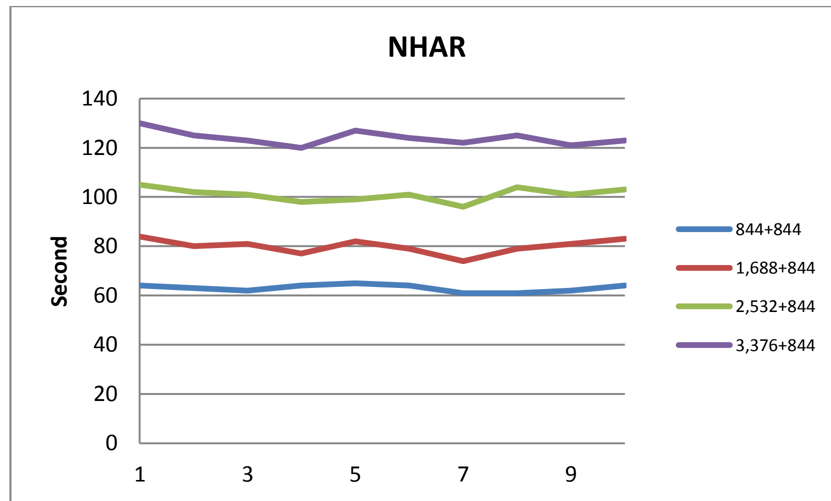
กราฟแสดงผลการทดลองการทำ Archive file

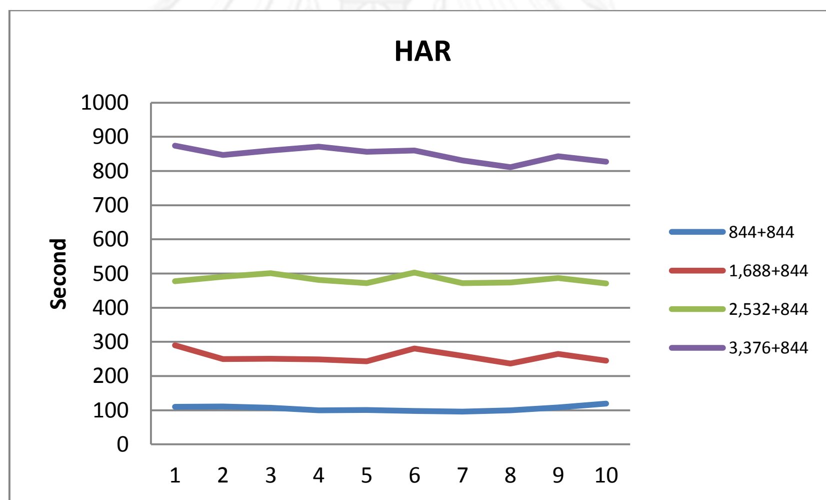




จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

กราฟแสดงผลการทดลองการเพิ่มไฟล์เข้าไปยัง Archive file





Chulalongkorn University

ประวัติผู้เขียนวิทยานิพนธ์

นางสาวจตุพร วรพงศ์กิติพันธ์ เกิดเมื่อวันที่ 7 กรกฎาคม พ.ศ. 2530 ที่จังหวัดกรุงเทพมหานคร สำเร็จการศึกษาระดับปริญญาบัณฑิต หลักสูตรวิศวกรรมศาสตรบัณฑิต สาขาวิชาเทคโนโลยีสารสนเทศ จากคณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง เมื่อปี พ.ศ. 2552 และเข้าศึกษาต่อในหลักสูตรวิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์ ณ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย เมื่อปี พ.ศ. 2554

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University

จุฬาลงกรณ์มหาวิทยาลัย
Chulalongkorn University