CMP 3

Preminilaries

Decomposition

↳ dividing a problem into smaller tasks

illustrated with

Dependency Graph

directed graph

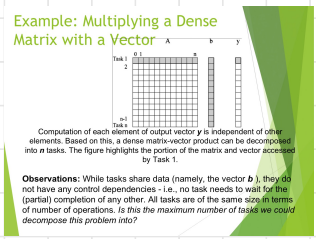
↳ nodes = tasks

↳ edges = dependency among tasks

Develop Parallel algo

reduces time to solve problem

1. Decompose problem into tasks that can be executed concurrently
2. Decomposed tasks can be of any size



Task Dependency Graph

node

↳ a task can only be executed when all tasks with incomming edges are completed

↳ nodes = tasks

↳ edges = dependency among tasks

↳ represents control dependencies

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Consider the execution of the query:

MODEL = ``CIVIC'' AND YEAR = 2001 AND
(COLOR = ``GREEN'' OR COLOR = ``WHITE)

decomposing query

Figure 3.2. The different tables and their dependencies in a query processing operation.

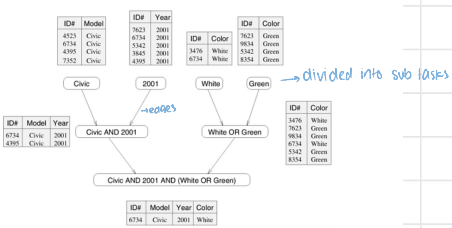
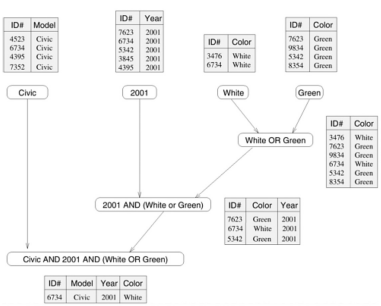


Figure 3.3. An alternate data-dependency graph for the query processing operation.



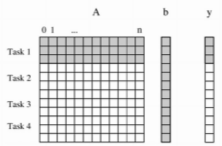
alternate decomposition may give significantly diff parallel performance

Granduality

↳ no of tasks into which a problem is decomposed

1. fine grained: decomposition into large no of tasks → increases speed up
2. coarse grained: decomposition into small no of tasks

Figure 3.4. Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.



→ coarse grained
1 task → 1, 2, 3 ?

Degree of concurrency

↳ no of tasks executed in parallel

↳ max d.o.c is the max no of tasks processed in parallel

↳ avg d.o.c is the avg no of tasks processed in parallel

↳ d.o.c increase as decomposition becomes finer in granduality

d.o.c: total work done / critical path length

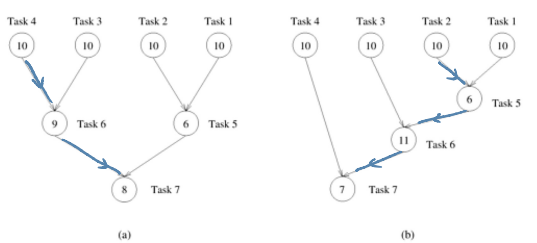
Critical path length

(sequence of tasks)

↳ length of longest path in task depedency graph

↳ longest path = shortest time for parallel execution

TASK DEPEDENCY GRAPH



	A	B
critical path length:	10+9+8 = 27	10+6+11+7 = 34
total work done:	10+10+10+10+6+9+8 = 63	10+10+10+10+6+11+7 = 64
avg d.o.c:	63/27 = 2.33	64/34 = 1.88

LIMITS ON PARALLEL PERFORMANCE

↳ innherent bound on how fine granduality can be

↳ communication overhead → due to concurrent tasks exchanging data with other tasks

↳ interaction
↳ data exchange

Task interaction graph

↳ decomposed sub tasks share data
↳ nodes = tasks
↳ edges = interactions
↳ represents data dependencies

Task dependency Graph

↳ a task can only be executed when all tasks with incomming edges are completed (node)
↳ nodes = tasks
↳ edges = dependency among tasks
↳ represents control dependencies

mapping

↳ mechanism by which tasks are assigned to processes (not Processors X — as we can combine tasks into a process)
↳ determined by both
↳ task depedency graph → ensure work is equally spread b/w Processes → max concurrency, optimal load balancing, min idlying
↳ task interaction graph → ensure processes need min interaction with eachother → min communication

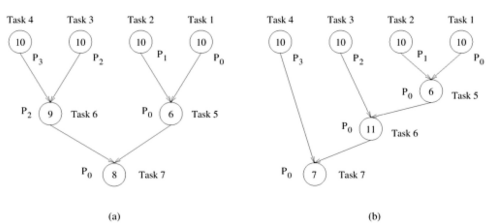
no of tasks in decomposition exceeds no of processing elements availabel → SOLUTION → mapping from tasks to Process

Mapping reduces Parallel execution time by

1. mapping independent tasks to diff processes → maximise use of concurrency
2. Assign tasks on critical path to processes as soon as they are available → minimize completion time
3. mapping tasks with dense (heavy) interactions to same process → minimize interaction b/w processes

(2 and 3 conflict with each other)

Figure 3.7. Mappings of the task graphs of Figure 3.5 onto four processes.



Mapping tasks in the database query decomposition to processes. These mappings were arrived at by viewing the dependency graph in terms of levels (no two nodes in a level have dependencies). Tasks within a single level are then assigned to different processes.

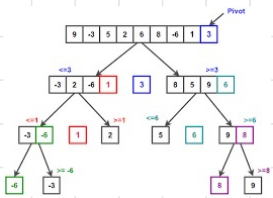
DECOMPOSITION TECHNIQUES

Recursive Decomposition

↳ divide and conquer strategy

1. decompose problems to sub problems
2. recursively decompose them further
3. until desired granularity reached

e.g Quick sort



Exploratory Decomposition

↳ investigate

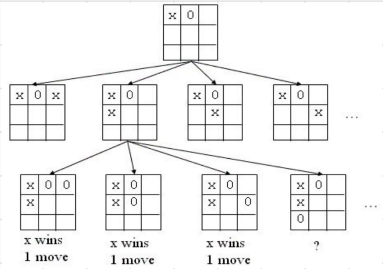
↳ generate all possibilities from start

↳ recursively check for best moves

↳ choose the found best move

e.g looking for best move in a game

- Simple case: generate all possible configurations from the starting position.
- Send each of the configurations to a child process.
- Each process will look for possible best moves for the opponent recursively eventually using more processes.
- When it finds the result, it sends it back to the parent. The parent selects the best move from all of the results received from the childs.



Speculative Decomposition

A classic example of speculative decomposition is in discrete event simulation.

- The central data structure in a discrete event simulation is a time-ordered event list.
- Events are extracted precisely in time order, processed, and if required, resulting events are inserted back into the event list.
- Consider your day today as a discrete event system - you get up, get ready, drive to work, work, eat lunch, work some more, drive back, eat dinner, and sleep.
- Each of these events may be processed independently, however, in driving to work, you might meet with an unfortunate accident and not get to work at all.
- Therefore, an optimistic scheduling of other events will have to be rolled back.

↳ anticipate possible computations

Data Decomposition

1. Identify data on which computations are performed
2. Partion it across various tasks

↳ Partition of the output across tasks decomposes the problem naturally

Consider the problem of multiplying two $n \times n$ matrices A and B to yield matrix C. The output matrix C can be partitioned into four tasks as follows:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

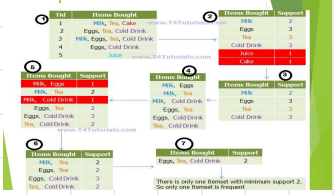
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem as in previus foil, with identical output data distribution, we can derive the following two (other) decompositions:

Decomposition I	Decomposition II
Task 1: $C_{1,1} = A_{1,1} B_{1,1}$	Task 1: $C_{1,1} = A_{1,1} B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$
Task 3: $C_{1,2} = A_{1,1} B_{1,2}$	Task 3: $C_{1,2} = A_{1,2} B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2} B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1} B_{1,2}$
Task 5: $C_{2,1} = A_{2,1} B_{1,1}$	Task 5: $C_{2,1} = A_{2,2} B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2} B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1} B_{1,1}$
Task 7: $C_{2,2} = A_{2,1} B_{1,2}$	Task 7: $C_{2,2} = A_{2,1} B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

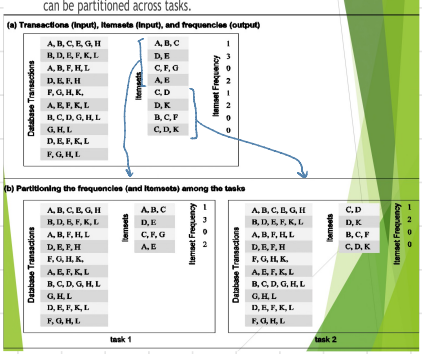
Data Decomposition: Example 2 -Frequent Itemset



Example 2 : discussion

- **Step 1:** Data in the database
- **Step 2:** Calculate the support/frequency of all items
- **Step 3:** Discard the items with minimum support less than 2
- **Step 4:** Combine two items
- **Step 5:** Calculate the support/frequency of all items
- **Step 6:** Discard the items with minimum support less than 2
- **Step 6.5:** Combine three items and calculate their support.
- **Step 7:** Discard the items with minimum support less than 2
- **Result:**
- Only one itemset is frequent (Eggs, Tea, Cold Drink) because this itemset has minimum support 2

Consider the problem of counting the instances of given itemsets in a database of transactions. In this case, the output (itemset frequencies) can be partitioned across tasks.



Output Data Decomposition: Example

From the previous example, the following observations can be made:

- If the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.
- If the database is partitioned across processes as well (for reasons of memory utilization), each task first computes partial counts. These counts are then aggregated at the appropriate task.

i) Input data decomposition is most suitable (see part b above) as it will reduce overall completion time as it calculates results for each chunk and combine them to produce the overall result.
ii) Output data decomposition is also possible; however, it either needs all data at each nodes responsible to compute each element of the output or additional communication with other nodes in order to fetch data elements stored there

ishma hafeez notes

6.1

Message Passing Paradigm

→ is both Asynchronous and loosely synchronous

↳ Parallel programing aproach

↳ It is characterized by 2 key attributes

↳ assumes a partitioned address space

↳ supports only explicit (not automatic) parallelization

↳ has additional hardware support for send and receiving messages

↳ may support DMA

↳ Asynchronous message transfer using Network interface hardware → receiver receive exactly the same message that was sent

↳ supports execution of different programs on each Processes

ADV

↳ Flexebility in Parallel Programing

DIS ADV

↳ writing parallel program becomes unscalable

↓ SOLUTION

Single Program Multiple data (SPMD)

→ is both Asynchronous and loosely synchronous

↳ most processes execute same code

Asynchronous

↳ all concurrent tasks execute non concurrently

↳ not coordinated

ADV

↳ can implement parallel also as each task is independent

DIS ADV

↳ harder to understand

↳ have non deterministic behavior due to race conditions

loosely synchronous

↳ tasks synchronize to perform interactions

↳ coordinated

ADV

↳ no race condition as dependent tasks → which subtask access first then which subtask ? "

DIS ADV

↳ individually tasks execute asynchronously

Message Passing Additional Hardware Support

DMA

↳ allows copying of data from 1 location to another without CPU support → once they are programmed

Asynchronous

both done using network interfaces

Network Interfaces

↳ allows transfer of messages from buffer to desired location without CPU intervention

6.2 Send and Read Operations

↳ used in message passing

```
1        P0                                P1
2
3        a = 100;                          receive(&a, 1, 0)
4        send(&a, 1, 1);                   printf("%d\n", a);
5        a=0;
```

(process 1 → send(&a, 1, 1); process 0 → receive(&a, 1, 0))

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

(sendbuf: buffer that stores data to be sent; nelems: no of data units sent and received; dest: process that receives data; recvbuf: buffer that stores data to be received; source: process that sends data)

1. P0 sends a=100 to P1
2. P1 receives in a and prints
3. if send is returned before P1 receives is executed → P0 changes value a:0 immediately after send
   ↳ P1 might receive 0

↓ SOLUTION

Blocking Message Passing Operations

↳ send operation to return only when semantically safe to do so
   ↳ sending operation blocked
   ↳ until no semantics will be violated on return
   ↳ irrespective of what happens in the program subsequently

↳ achived by 2 mechanisms
1. Blocking non buffered
2. Blocking buffered

1. BLOCKING NON BUFFERED S/R → does idling

↳ sending operation does not return until the matching receive has received first

↳ Typically it involves a handshake b/w sending and receiving → *since it doesn't use any buffer hence also non blocking buffer operation

↳ First sending process sends a request to communicate to the receiving process

↳ receiving then responds to the request

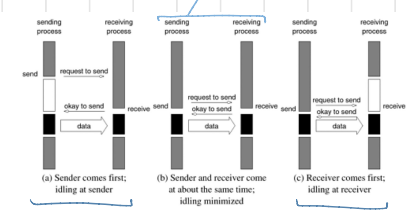
↳ then the sender initiates a transfer

CONS

↳ idling overhead → fixed when both come at the same time → but hard to predict for asynchronous

↳ deadlocking → receive calls are always blocking

blocking non buffer scenario




```
1          P0                          P1
2
3          send(&a, 1, 1);             send(&a, 1, 0);
4          receive(&b, 1, 1);          receive(&b, 1, 0);
```

↳ P0 and P1 both access a
↳ using blocking non buffer protocol
↳ send at P0 waits for matching receive at P1
↳ while send at P1 waits for matching receive at P0
↳ resulting in infinite loop

idling overheads

Deadlock

SOLUTION

BLOCKING BUFFERED S/R

2. BLOCKING BUFFERED S/R

↳ does buffer management

↳ solution for idlying overhead and deadlock → some deadlock still possible (although many alleviated)

↳ caused only by waits in receive operations

```
1        P0                          P1
2
3        receive(&a, 1, 1);          receive(&a, 1, 0);
4        send(&b, 1, 1);             send(&b, 1, 0);
```

↳ both processes wait to receive data but nobody sends it

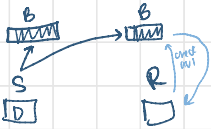
↳ rely on buffers for sending and receiving

↳ sender has a preallocated buffer

↳ sender copies data into buffer

↳ returns after copy operation is completed

↳ This communication can be done many ways



↳ 1. If hardware supports asynchronus communication → CPU independent

↳ then network transfer can be initiated

↳ after message has been copied to buffer

• at receiving end data cant directly be stored at target location (as violates program semantics)

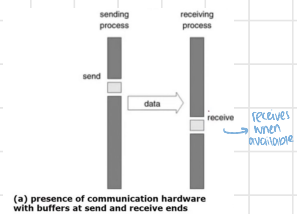
STEPS

↳ instead data is copied into buffer at receiver as well

↳ when receiving process encounters receive operation

↳ it checks if message is avl in its receive buffer

↳ if so, data is copied into target location

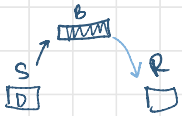


(a) presence of communication hardware with buffers at send and receive ends

↳ 2. If no dedicated hardware (saves some of the overhead)

↳ then buffering on one side at sender and

↳ receiver initiates tranfer by interrupting sender



STEPS

↳ on encountring a send operation

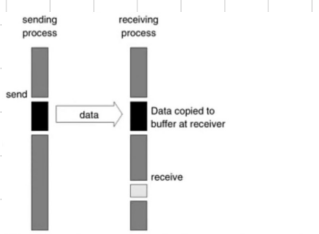
↳ the sender interupts the receiver

↳ both process participate in comm. operation and

↳ message is deposited at the receiver end

↳ when receiver encounters receive operation

↳ the data is copied from buffer to target location



(b) absence of communication hardware, sender interrupts receiver & deposits data in buffer at receiver end

Finite buffers in message passing impact

```
1
2
3
4
5
6
```

```
P0

for (i = 0; i < 1000; i++) {
  produce_data(&a);
  send(&a, 1, 1);
}
```

```
P1

for (i = 0; i < 1000; i++) {
  receive(&a, 1, 0);
  consume_data(&a);
}
```

CONS
↳ unforseen overheads
↳ performance degradation

↳ P0 produces 100 data items, P1 consumes them
  ↳ If P1 was slow then
    P0 might have sent all its data
  ↳ if there isn't enough buffer space then → buffer overflow
    sender will be blocked until
    some of the corresponding read operations are posted
    thus freeing buffer space

Non Blocking message passing operations

↳ returns send/receive operation before semantically safe to do so
↳ generally have a check status operation — indicates whether semantics of a previously initiated transfer is violated or not
↳ when theres a return from send/receive operation
  ↳ process can perform any computation
    ↳ that doesnt involve completion of operation
↳ process waits for completion of operation

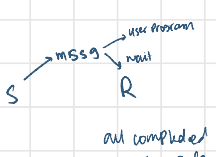
PROS
↳ fast send/receive
↳ incurs little overhead
↳ idling time is utilised for computation

It can be buffered or non buffered

Non Buffered

↳ a process wishing to send data to another can
↳ post a pending message
↳ and return to user program — so program can do other useful work
↳ when corresponding receive is posted
  ↳ comm. operation is initiated
↳ when operation completed
  ↳ check status operation indicates wether safe for programmer to touch this data

eg 6.4 a)



Buffered

↳ sender initiates DMA operation and returns immediately
  ↳ the data becomes safe when DMA operation completed
↳ the receive operation initiates a transfer
  ↳ from senders buffer to target location

PROS
↳ reduces time when data is safe

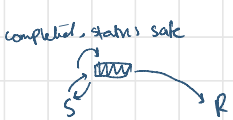


Figure 6.4. Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

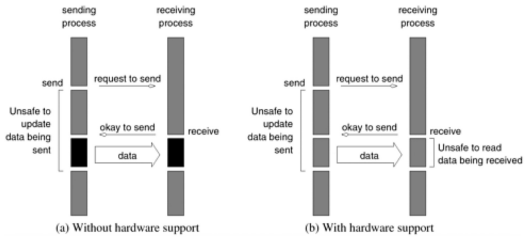
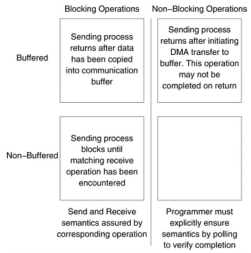


Figure 6.3. Space of possible protocols for send and receive operations.



communication hardware. This is illustrated in Figure 6.4(b) . In this case, the communication overhead can be almost entirely masked by non-blocking operations. In this case, however, the data being received is unsafe for the duration of the receive operation.

BLOCKING VS NON BLOCKING

↳ safe programming

↳ easier programming

↳ performance optimization → by masking comm. overhead

CONS

↳ errors can result in unsafe access to data that is process of being communicated

Hadoop Distributed File System (HDFS)

↳ stores large data sets reliably

↳ streams those data at high bandwith → by replicating file content on multiple DNs

↳ fast access to meta data → as entire meta data is stored in RAM

BackUp Node · HDFS client · Data Node · CheckPoint Node · Snapshots

Architecture

↳ files split into large blocks → typically 128 kB

↳ blocks are stored across DNs

↳ each block is duplicated at multiple Data Nodes → typically 3 → to prevent data loss

Name Node

aka master

↳ stores metadata → e.g. File data, Permissions, Modifications, Namespace, Size; location, No of replications, Rackid, Your business location

↳ receives heart beats and block report from DN

↳ maintains latest metada from block reports

↳ there are 2 files associated with metadata

updates

Image

↳ contains complete state of file system namespace → e.g. which file is where

↳ record of image stored locally is a checkpoint

↳ checkPoint is never changed by NN it is replaced entirely when new check point is created → during restart

updates

Journal

↳ contains all recent modification log of image → e.g. deleted, changed location

↳ updates image file of all changes

↳ every change by client is recorded in the journal

↳ journal is flushed and synced then the change is commited by client

NN During start up

↳ NN initializes image from checkpoint

↳ updates changes from journal till image up-to-date

↳ a new checkpoint and empty journal are written back before NN starts serving clients

Operations

↳ file read/write

↳ Block Placement

↳ Replica management

↳ Balancer

Data Node → actually stored here

↳ Physical location of file data

↳ performs read/write requests

↳ performs block deletion, creation, replication upon instruction from NN

↳ sends block reports to NN periodically

gives the specifications of journal and image

Single Name Node

↳ maintains namespace tree operations → hierarchy of files and directories → e.g. closing and renaming file directories

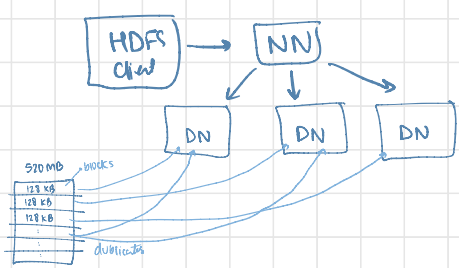
↳ maintains mappings of file blocks to DN

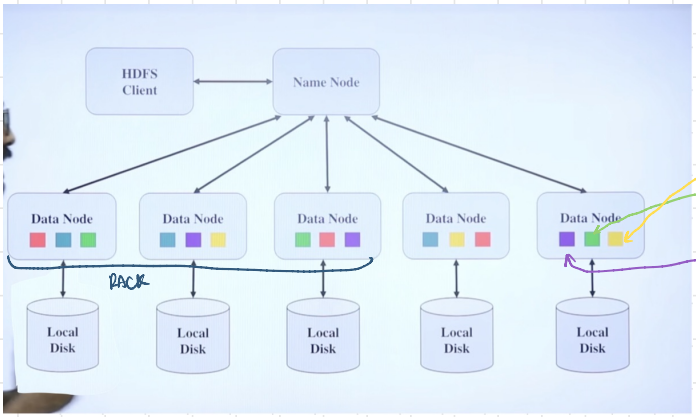
↳ File metadata → inode

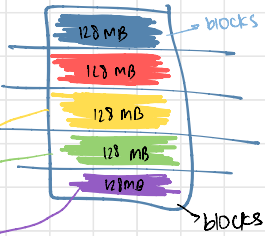
↳ authorization and authentication

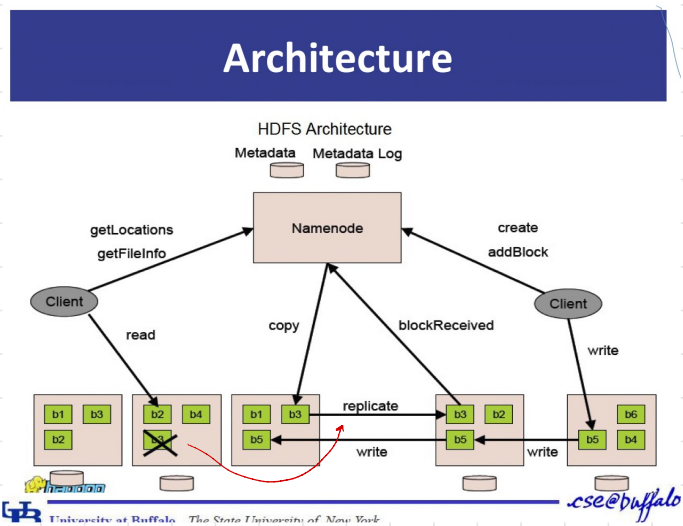
↳ collect Block reports from DNs

↳ Replicate missing Blocks









HDFS Adv

- ↳ cost effective → same hardware
- ↳ fault tolerance → if 1 data node is down a copy still exists → each data block is replicated and distributed across diff DNs (default 3 times)
- ↳ gives easy access
- ↳ maintain system integrity → Heart Beat
- ↳ fast access to metadata → entire namespace is kept in ram

used by NN for NN's space allocation and load balancing decisions

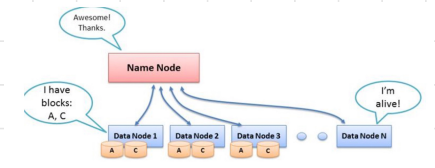
contain info about
- total storage capacity
- fraction of storage in use
- no. of data transfers currently in process

Heart Beats

↳ NN and DN communicate by heartbeats

↳ DN sends HB to NN every 3 sec to confirm DN is operating and block replicas it hosts are available

↳ NN uses replies to HB to send instructions to DN



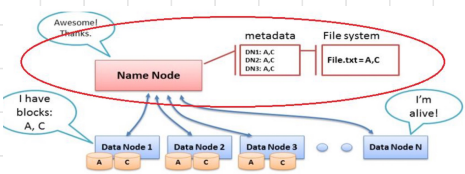
Block Report

↳ DN sends BR to NN to report block replicas in its possession

↳ keep NN up-to-date to where block replicas are located in the cluster

↳ every 10th heartbeat is a block report

↳ It contains
- ↳ block id
- ↳ generation stamp
- ↳ block replica length



Re-replicating missing replicas

HB fail

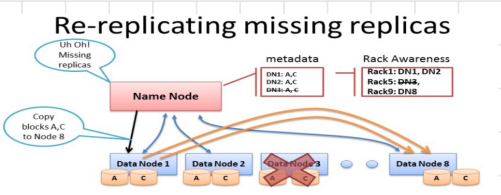
↳ If NN doesn't receive a HB in 10 mins

↳ It signifies lost DN

↳ block replicas hosted by DN become unavailable

↳ NN consults Rack Awareness Script

↳ NN schedules creation of new replicas of those block on other DNs



- Missing Heartbeats signify lost Nodes
- Name Node consults metadata, finds affected data
- Name Node consults Rack Awareness script
- Name Node tells a Data Node to re-replicate

TWO OTHER ROLES OF NN

Check Point Node

- local storage of image
- when journal (change log) becomes too long
- CN combines existing CN and journal to create a new CP and empty journal

Back UP NODE

- read only NN
- maintains an in-memory
- updates image of the file
- if NN fails, BN's image in memory and the check point on a disk is a record of latest namespace state

Snapshots

- minimizes potential damage to data stored during upgrades
- SS persistently saves current state of file system,
- if upgrade results in loss/data corruption, its possible to roll back upgrade and return HDFS to the namespace and storage state as in SS

NN and DN During start up

- each DN connects to NN and
- performs a handshake to verify
  - namespace id
  - software version of DN

(If either doesn't match DN shuts down)

- DN registers with NN
- DN stores their unique storage id (internal identifier of DN doesn't change)

Rack Awareness

- physical collection of various DN
- never loose all data if entire rack fails
- choosing closes data node for serving a purpose

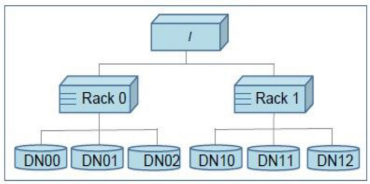


Figure 3. Cluster topology example

Replica placement policy

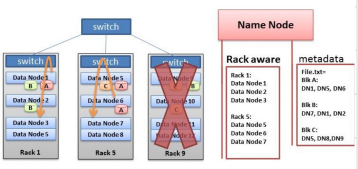
- no DN contains > 1 replica of any block
- no rack contains > 2 replicas of same block (provided there are sufficient racks on the cluster)

NN avoid replicas of block at same rack

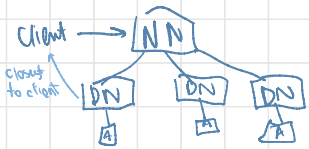
(collection of DN)

- if a DN identifies block replicas
  - it sends a block report to NN
- if NN detects a replicas of blocks at one rack
  - the block are treated as under-replicated
  - and replicates block to a diff rack
  - then old replica is deleted (to avoid over replication)

Hadoop Rack Awareness – Why?



Never loose all data if entire rack fails



HDFS client wants to read a file

↳ client contacts NN to get list of DN's hosting replicas of the blocks of file

↳ NN requests DN to transfer desired block

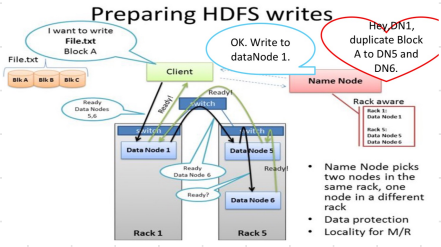
↳ reads block contents from DN closest to client

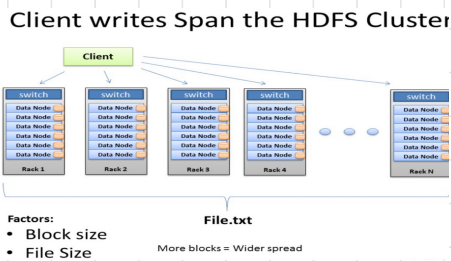
HDFS client wants to write to a file

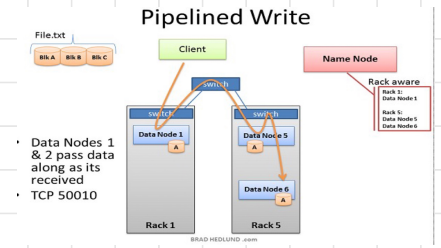
↳ client asks NN to choose 3 DN that host block replicates

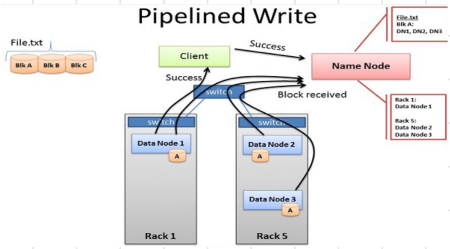
↳ client writes data to DN's in pipeline fashion

the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of DataNodes is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in









ishma hafeez
notes