Corso di Laurea triennale

in INGEGNERIA INFORMATICA

Tesina di Laurea

Il backup incrementale di una working directory

di Guido Panzetta

Anno Accademico 2020/2021

Indice

Il backup incrementale

Il linguaggio dell’applicazione

Prima esecuzione: creazione dell’ambiente di lavoro e del dizionario

Successive esecuzioni dell’applicazione

Esplorazione della working directory

Termine dell’esecuzione

Utilità dell’applicazione

Tempo di esecuzione

Caratteristiche della working directory

Simulazioni relative al primo lancio dell’applicazione: il backup totale

Simulazioni relative a lanci successivi dell’applicazione: il backup incrementale

Conclusioni sull’utilità dell’applicazione

Repository e GitHub

**IL BACKUP INCREMENTALE DI UNA WORKING DIRECTORY**

Un progetto software, sviluppato da un gruppo di lavoro, è solitamente composto da molti moduli, librerie e centinaia di programmi. Necessita di continui aggiornamenti che devono essere integrati. A volte questo processo risulta assai complesso. Diventa indispensabile effettuare regolarmente il backup ovvero la duplicazione del contenuto di un sistema informatico memorizzato su un supporto esterno (disco rigido o chiavetta USB), in un server **NAS (**Network Attached Storage), cloud oppure online.

Tra le varie tipologie, il backup incrementale utilizza una precedente copia del sistema, completa o incrementale e va a duplicare e salvare solo i file nuovi e quelli modificati. Lo spazio richiesto è minore, il salvataggio dei dati diviene facile da gestire e l’aggiornamento risulta veloce.

In tal senso ho realizzato un’applicazione che effettua il backup incrementale di una working directory.

**IL LINGUAGGIO DI PROGRAMMAZIONE DELL’APPLICAZIONE**

Il linguaggio di programmazione utilizzato per l’algoritmo è il **Python**, che mette a disposizione, tra i vari tipi di dato, i ***dizionari***, sequenze di elementi caratterizzati da coppie chiave-valore.

L’idea alla base del progetto è stata generare, al lancio dell’applicazione, un dizionario (*index*) che contenesse come chiavi le absolute path dei file che fanno parte della working directory e come valori associati i relativi rifermenti temporali di creazione e modifica. Si vuole verificare se l’utilizzo del dizionario in memoria consenta di individuare più rapidamente i file da salvare rispetto al normale accesso e controllo su disco locale. La sua utilità verrà discussa nel seguito.

Vediamo meglio quale sia il funzionamento e la logica dell’applicazione implementata in dizbackup.gy.

**PRIMA ESECUZIONE: CREAZIONE DELL’AMBIENTE DI LAVORO E DEL PRIMO DIZIONARIO**

La prima volta che si esegue il codice e si esplora la working directory sono creati:

* una ***Backup\_Directory*** locale, ove è riprodotta la struttura ad albero della *Working\_Directory* ed ivi copiati tutti i file in essa contenuti, nelle rispettive sottocartelle. Inizialmente viene prodotto un backup totale del sistema. La prima esecuzione pertanto è molto più lenta delle successive per le numerose istruzioni di copy necessarie[[1]](#footnote-1).
* un **dizionario** (*index*) che ha per chiavi le absolute path dei file che compongono la working directory e per valori i riferimenti temporali di creazione e modifica. I dati ivi contenuti vengono scritti in un file *(index.gz),* compresso e serializzato all’interno della Backup Directory in locale.

**SUCCESSIVE ESECUZIONI DELL’APPLICAZIONE**

Ad ogni lancio successivo dell’applicazione, al momento di effettuare un nuovo backup incrementale, il file index.gz, che si trova nella *Backup\_Directory*, viene aperto in lettura e scompattato[[2]](#footnote-2). L’algoritmo genera e carica in memoria, con i dati letti dal file, un dizionario composto da elementi chiave-valore (le absolute path dei file della *Working\_Directory* ed i relativi riferimenti temporali di creazione e modifica che questi avevano alla data del precedente backup[[3]](#footnote-3)).

**ESPLORAZIONE DELLA WORKING DIRECTORY**

Durante l’esecuzione dell’algoritmo viene esplorata l’intera working directory. Nelle varie sottocartelle si incontreranno probabilmente nuovi file, oltre a quelli precedentemente progettati e già oggetto di backup. Taluni di questi ultimi, durante lo sviluppo di un software, possono essere stati modificati ed avere quindi dei riferimenti temporali (i c.d. *metadati*) successivi rispetto a quelli dei medesimi file memorizzati nell’ultimo backup e, di conseguenza, anche rispetto ai valori delle chiavi del dizionario index, costruito in memoria con i dati del file index.gz, serializzato alla data del precedente salvataggio.

L’algoritmo, pertanto, verifica che ogni file incontrato nell’esplorazione della working directory sia presente come chiave (la absolute path del file) del dizionario in memoria:

* **in caso negativo** il file è **NUOVO** e viene inserito, come elemento chiave-valore, nel dizionario attualmente in memoria e copiato nella backup directory, in posizione corretta.
* **in caso positivo** il file **NON È NUOVO** e nella cartella di backup ne esiste uno con la stessa absolute path. Per verificare se il file incontrato sia stato modificato dall’ultimo backup, si confronta la data di ultima modifica con il valore corrispondente alla chiave presente nel dizionario in memoria[[4]](#footnote-4). In caso di modifica dall’ultimo backup il riferimento temporale presente nel dizionario (il valore della chiave) sarà antecedente e dovrà essere aggiornato con quello attuale. Copia dell’ultima versione del file deve essere memorizzata nella backup directory[[5]](#footnote-5) in posizione corretta.

**TERMINE DELL’ESECUZIONE**

Prima del termine dell’esecuzione gli elementi del dizionario (le chiavi e i valori associati aggiornati alla data attuale) vengono scritti nel nuovo file index.gz[[6]](#footnote-6), che viene ancora una volta compattato e serializzato[[7]](#footnote-7) e reso disponibile per un futuro backup.

Alla fine dell’esecuzione la backup directory è aggiornata e contiene i vecchi file non oggetto di revisione nel periodo tra due backup successivi, i nuovi inseriti e quelli modificati in tale arco temporale.

**UTILITÀ DELL’APPLICAZIONE**

L’utilità dell’applicazione può essere individuata nella rapidità di esecuzione del backup di una working directory: l’utilizzo del dizionario, caricato in memoria con i dati del file index.gz, dovrebbe teoricamente rendere più rapido il riconoscimento dei file nuovi e di quelli modificati rispetto al controllo tramite accesso su disco, seppur in locale.

**TEMPO DI ESECUZIONE**

Il tempo reale di esecuzione di un algoritmo è fortemente influenzata dalle capacità computazionali della macchina (computer) in cui viene eseguito l'algoritmo[[8]](#footnote-8) e dagli interventi del sistema operativo durante tale attività. Ciò rende particolarmente difficile il confronto omogeneo delle performance tra algoritmi diversi. Per questa ragione si preferisce solitamente calcolare una durata di esecuzione “teorica”, fissando il valore temporale di ogni istruzione[[9]](#footnote-9).

Ciò premesso, sono state effettuate delle simulazioni di esecuzione dell’applicazione per verificare la concreta utilità dell’algoritmo progettato e appurare se l’utilizzo del dizionario abbia fornito concretamente una maggiore rapidità nella selezione dei file nuovi e modificati rispetto all’accesso su disco in locale.

A tal fine sono stati utilizzati due algoritmi simili: “dizbackup” e “nodizbackup”. Il numero delle istruzioni del nucleo centrale[[10]](#footnote-10) di entrambi può considerarsi pressoché equivalente e la differenza risiede nelle modalità di ricerca e controllo dei file nuovi o modificati: dizbackup utilizza il dizionario, mentre nodizbackup effettua la ricerca e la verifica della data di ultima modifica con accesso su disco locale, in particolare nella cartella di backup.

Per misurare il tempo di esecuzione[[11]](#footnote-11) è stato inserito un timer (start and stop sotto riportato), limitato al solo blocco di istruzioni che effettua, in ciascun algoritmo, questo controllo[[12]](#footnote-12).

start = time.time()

stop = time.time()

durata = end – stop

**CARATTERISTICHE DELLA WORKING DIRECTORY**

Sono state effettuate un centinaio di simulazioni utilizzando una working directory articolata, contenente 744 sub-directory e 6432 file.

**SIMULAZIONI RELATIVE AL PRIMO LANCIO DELL’APPLICAZIONE: IL BACKUP TOTALE**

Alla prima[[13]](#footnote-13) esecuzione, in assenza di un precedente salvataggio dei dati, i due algoritmi effettuano il backup totale della working directory, creando la backup directory e la relativa struttura ad albero.

Per ogni file incontrato nell’esplorazione della directory:

* *dizbackup* inserisce nel dizionario appena inizializzato e privo di elementi, la chiave (la absolute path) e il relativo valore (la data di creazione e modifica) del file e ne effettua la copy nella backup directory.
* *nodizbackup*, invece, verifica che questo non sia presente nella cartella di backup poco prima creata e quindi “vuota” e successivamente lo copia.

A causa delle molte istruzioni di copy questa attività risulta dispendiosa in termini di tempo.

**Miglior tempo dizbackup: 161.93 secondi**

**Miglior tempo nodizbackup: 176.38 secondi**

I tempi ottenuti mostrato che, la prima volta che si esegue l’algoritmo e si effettua il backup totale del sistema, l’utilizzo del dizionario riduce il tempo di esecuzione del 8% circa[[14]](#footnote-14) rispetto a quello ottenuto dall’algoritmo nodizbackup che opera il controllo su disco locale.

**SIMULAZIONI RELATIVE A LANCI SUCCESSIVI DELL’APPLICAZIONE: IL BACKUP INCREMENTALE**

**CASO MIGLIORE:** non è stato inserito alcun nuovo file e non sono stati modificati quelli già presenti nella working directory.

I due algoritmi, nell’esplorazione della working directory, non effettuano alcuna istruzione di copy limitandosi al controllo dell’esistenza e della data di modifica dei file incontrati.

**Miglior tempo dizbackup: 0.56 secondi**

**Miglior tempo nodizbackup: 1.43 secondi**

In questa situazione il dizionario in memoria riduce notevolmente il tempo di esecuzione dell’algoritmo, sebbene in entrambi sia molto breve poiché, come detto, non si effettua alcuna copy nella cartella di backup.

**CASI INTERMEDI**

1. **NUMEROSI FILE MODIFICATI DALL’ULTIMO BACKUP**

Nella working directory sono stati inseriti 100 nuovi file e 4165 sono stati modificati[[15]](#footnote-15).

Le simulazioni[[16]](#footnote-16) relative ad esecuzioni successive alla prima dei due algoritmi hanno mostrato, con questo tipo di directory, che i tempi di esecuzione dei due algoritmi sono pressoché equivalenti.

Il dizionario, pur dovendo teoricamente ridurre il tempo di esecuzione dell’algoritmo rispetto a nodizbackup che opera su disco locale, non fornisce un apprezzabile miglioramento.

Gli interventi del sistema operativo in corso di esecuzione e la necessità di aggiornare le chiavi del dizionario, non prevista nel secondo, equilibrano i risultati ottenuti.

**Miglior tempo di backup incrementale ottenuto da dizbackup: 45.81**

**Miglior tempo di backup incrementale ottenuto da nodizbackup: 45.61**

Nelle esecuzioni successive alla prima e con un numero elevato di file modificati l’utilità dell’algoritmo appare limitata.

1. **POCHI FILE MODIFICATI DALL’ULTIMO BACKUP**

All’interno della working directory sono stati inseriti 10 nuovi file e solo 100 sono stati modificati[[17]](#footnote-17).

**Miglior tempo di backup incrementale ottenuto da dizbackup: 3.43**

**Miglior tempo di backup incrementale ottenuto da nodizbackup: 4.32**

Nelle esecuzioni successive alla prima, con un numero limitato di file modificati, l’utilità dell’algoritmo si ritiene dimostrata.

**CASO PEGGIORE:** tutti i file della working directory sono stati modificati.

Questa condizione è analoga alla prima esecuzione dell’applicazione, quando tutti i file della working directory devono essere copiati.

Il tempo di esecuzione dei due algoritmi corrisponde sostanzialmente a quello necessario per effettuare il backup totale.

**CONCLUSONI SULL’UTILITÀ DELL’APPLICAZIONE**

Dalle simulazioni si può concludere che l’utilizzo del dizionario in memoria riduce lievemente[[18]](#footnote-18) il tempo di esecuzione del backup e l’applicazione può considerarsi utile.

**REPOSITORY SU GITHUB**

Uno strumento ormai indispensabile per gestire il c.d. *versioning* è il repository[[19]](#footnote-19).

GitHub, come altri analoghi strumenti, è un hosting che offre uno spazio gratuito fondamentale per gestire al meglio progetti per lo sviluppo di software condiviso con altri collaboratori. Ad ogni “commit” e “push” in aggiornamento dei programmi o dei singoli moduli segue la “pull” degli altri collaboratori, che possono operare anche da remoto e integrare i vari moduli nel progetto.

1. Shutil.copy2(sorgente, destinazione)). L’istruzione copy2 salva anche i metadati nel file di destinazione. [↑](#footnote-ref-1)
2. index = pickle.load(gzip.open(indexFile, "rb")) [↑](#footnote-ref-2)
3. Se fosse la seconda esecuzione dell’algoritmo il dizionario conterrebbe le absolute path e i riferimenti temporali di creazione e modifica dei file che memorizzati nel backup totale effettuato la prima volta. Se fosse una successiva esecuzione il dizionario conterrebbe le absolute path e i riferimenti temporali dei file che compongono la working directory alla data del precedente backup incrementale. [↑](#footnote-ref-3)
4. Che avrà i riferimenti temporali di ultima modifica dello stesso file alla data del precedente backup. [↑](#footnote-ref-4)
5. Nella Backup Directory l’ultima versione del file deve sostituire quella precedente. [↑](#footnote-ref-5)
6. Che va a sostituire il vecchio file [↑](#footnote-ref-6)
7. pickle.dump(index, gzip.open(indexFile, "wb")) [↑](#footnote-ref-7)
8. Il pc utilizzato è un HP Intel core i7 del 2017, ormai superato. [↑](#footnote-ref-8)
9. Ad esempio, può essere attribuito un tempo standard pari a 1 microsecondo per ogni passo elementare elaborato dagli algoritmi a parità di dimensione dei dati in input. [↑](#footnote-ref-9)
10. Le istruzioni che effettuano la ricerca e verifica dei file nuovi o modificati nelle due differenti modalità. [↑](#footnote-ref-10)
11. Come premesso, i risultati delle simulazioni variano sostanzialmente dal pc utilizzato e dall’attività del sistema operativo durante l’esecuzione dell’algoritmo. [↑](#footnote-ref-11)
12. Sono state escluse dalla misurazione del tempo di esecuzione le istruzioni di accesso in lettura al file index.gz e la sua successiva serializzazione dopo l’aggiornamento del dizionario. [↑](#footnote-ref-12)
13. La prima esecuzione crea il backup totale della working directory e può risultare dispendiosa in termini di tempo a causa delle numerose istruzioni di copy effettuate. [↑](#footnote-ref-13)
14. I risultati sono ritardati da interventi del sistema operativo. [↑](#footnote-ref-14)
15. È stato utilizzato un algoritmo che inserisce il carattere “a” in coda ai file, in modo che abbiano la data di modifica posteriore rispetto a quella presente nel file index.gz e nella backup directory. [↑](#footnote-ref-15)
16. Il sistema operativo interviene e spesso altera i risultati. [↑](#footnote-ref-16)
17. È stato utilizzato un algoritmo che inserisce il carattere “a” in coda al file. In tal modo i 1000 file hanno la data di modifica posteriore rispetto a quella presente nel file index.gz e nella backup directory. [↑](#footnote-ref-17)
18. Il calcolo esatto è di difficile rilevazione. [↑](#footnote-ref-18)
19. Da Wikipedia: un repository (letteralmente deposito o ripostiglio), in informatica, è un ambiente di un sistema informativo (ad esempio di tipo ERP), in cui vengono gestiti i metadati, attraverso tabelle relazionali; l'insieme di tabelle, regole e motori di calcolo tramite cui si gestiscono i metadati prende il nome di metabase. [↑](#footnote-ref-19)