

Appunti Basi di Dati e Web - Teoria

Organizzazione fisica e gestione delle interrogazioni

Ci concentriamo ora sul capitolo 11 del libro.

Abbiamo imparato a usare i servizi del DBMS come se questo fosse una scatola nera, quindi senza guardare cosa c'è dentro, cosa succede all'interno del DBMS.

In questo corso cerchiamo di capire come funziona il DBMS, qual è la sua architettura e quali sono le sue caratteristiche.

Perché?

Per conoscerlo meglio e per sapere quali servizi sono offerti e come

Data Base Management System



Definizione

È un sistema informativo, in grado di gestire una collezione di dati.

Caratteristiche dei dati:

- Grande

I dati raccolti crescono sempre nel tempo. La dimensione di una base di dati è maggiore dei sistemi che la utilizzano. Devo pensare di memorizzare la base di dati in una memoria secondaria, anche questo riguarda la persistenza.

- Persistente

Il dato è persistente rispetto alla singola esecuzione del programma. Questo riguarda anche i guasti: ne devo garantire l'usabilità a fronte dei guasti.

- Condivisa

Posso pensare che utenti diversi utilizzino parti diverse della stessa base di dati. Devo far in modo di poter distinguere rispetto a chi accede a quale parte e come vi accede.

Garanzie del DBMS:

- Affidabilità

Non perdo ciò che ho fatto alla fine di un'operazione.

- Privatezza

Devo controllare chi accede a cosa.

- Efficiente

- Efficace

Le basi di dati sono grandi e persistenti

Ho necessità di una memoria secondaria, la dimensione richiede una memoria esterna a quella principale, che riguardi solo il mantenimento dei miei dati. La grandezza dei dati richiede anche che tale che la

gestione sia sofisticata.

Si comporta analogamente a un sistema operativo quando gestisce la memoria.

Architettura a livelli

I DBMS sono organizzati su vari livelli:

- Livello LOGICO: a cui fanno riferimento gli utenti (SQL, di cui abbiamo già parlato)
- Livello FISICO: stabilisce l'effettiva implementazione della memorizzazione di quello che abbiamo progettato.

Ad esempio la progettazione è: concettuale → logica → fisica, durante il corso di basi di dati ci siamo soffermati solo sul modello concettuale e logico, non siamo scesi a livello fisico.

L'architettura di un DBMS prevede il fatto che le tabelle vengano realizzate utilizzando diverse strutture fisiche. Quali sono queste strutture? Risponderemo a questa domanda.

In generale la proprietà di **INDIPENDENZA** dei dati, garantisce che gli utenti possano ignorare le strutture fisiche, posso usare la base di dati non sapendo tutto quello che ci sta sotto.

Questa proprietà ha reso il modello relazionale molto importante ed usato.

- In che modo sono organizzati i dati?
- In che modo il DBMS gestisce le interrogazioni?

Nel momento in cui costruisco la base di dati, penso già al modo in cui questa verrà interrogata tramite l'utente, secondo un modello logico.

I dati sono quindi in memoria secondaria → ho bisogno di strutture fisiche opportune per memorizzare.

La memoria secondaria è più lenta della memoria principale: devo far interagire una cosa molto veloce con una cosa un po' più lenta. Tanto più riesco a limitare, e ottimizzare, quello che faccio tanto più il DBMS sarà efficiente.

Componenti coinvolte in un DBMS

Nel dettaglio:

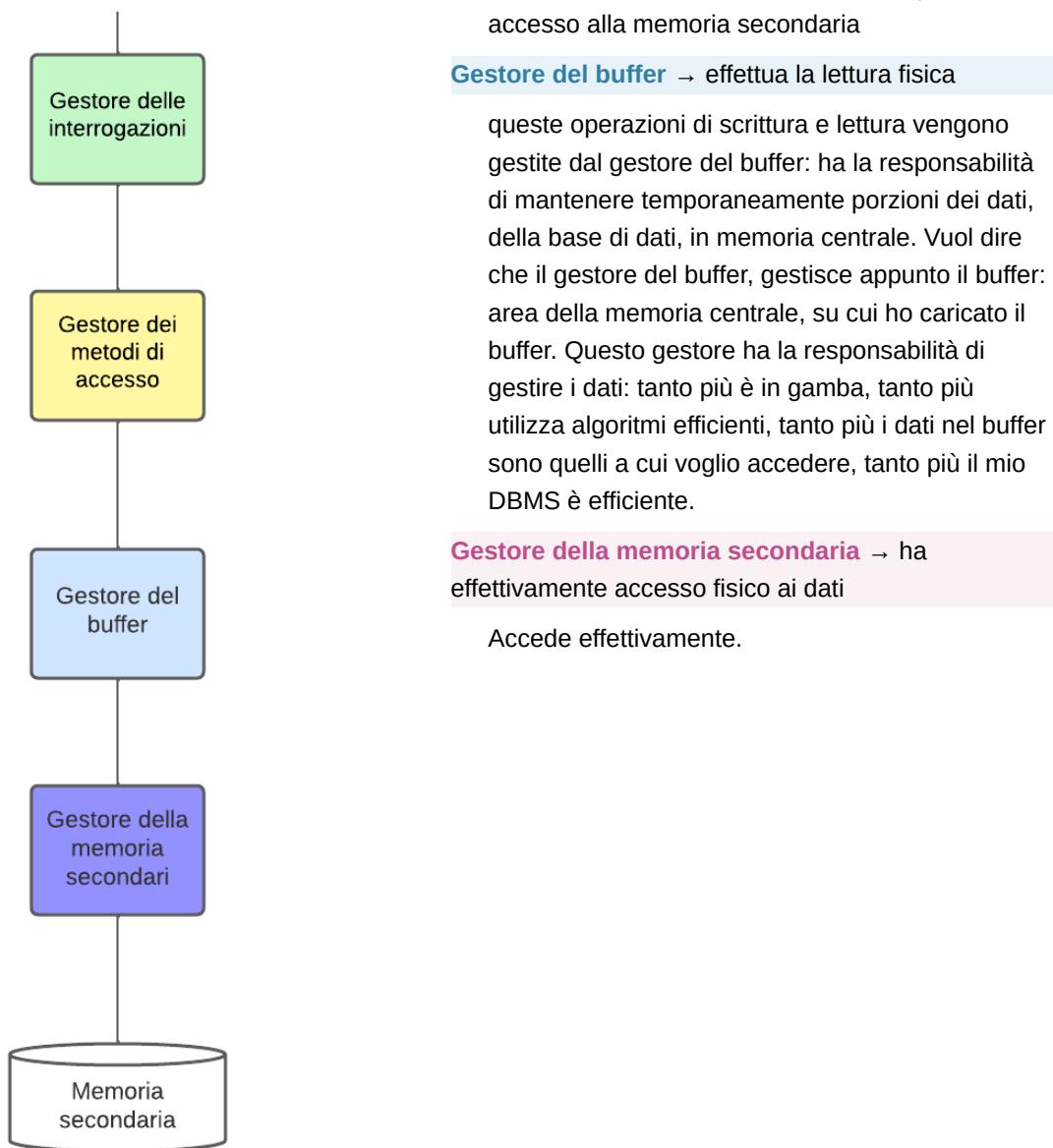
Gestione delle interrogazioni → scansione, accesso diretto, ordinamento: scomponere la query SQL in operazioni primarie

Prende in input una interrogazione in SQL e produce una sorta di trasformazione in termini operazioni di basso livello (scansione, accesso diretto, ordinamento)

Questo insieme di operazioni vengono trasmesse al gestore successivo:

Gestione dei metodi di accesso → cerca di capire ai dati a cui devo accedere

Conosce i dettagli della struttura fisica, sa come sono memorizzati i miei dati a livello fisico. Per gestire le operazioni di basso livello le trasforma in



Domanda:

Sia dia una descrizione, si costruisca lo schema dei componenti del DBMS, dandone una breve descrizione.

Affidabilità

Dalle cose più stupide, a quelle più catastrofiche, devo garantire che se gli succedesse qualcosa (livello software, ma anche livello hardware) non posso perdere neanche una tupla.

Le *transazioni* devono essere:

- **atomiche**

O vengono portate a buon fine, o nulla: non c'è un intermezzo. O tutto o niente.

- **definitive**

Non importa quando, ma una volta fatta e conclusa, rimane tale

Questo è impegnativo a causa di:

- Aggiornamenti frequenti
- Necessità di gestire il buffer

Condivisione

Una base di dati è una risorsa integrata, condivisa fra le varie applicazioni.

Conseguenze:

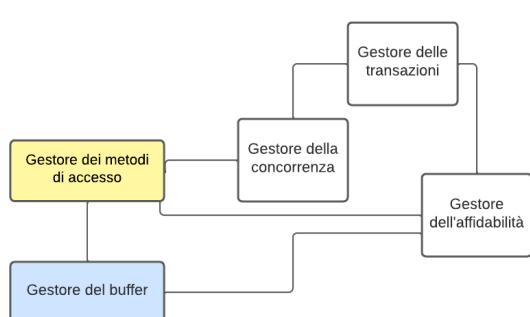
- Meccanismi di autorizzazione

Chi può accedere a cosa

- Meccanismi di concorrenza

Attività multiutente su dati condivisi

Il controllo della concorrenza fa sì che io non debba avere solo **transazioni seriali** (che limiterebbe e renderebbe il mio sistema meno efficiente): quello che faccio è simulare la serialità, in realtà sto gestendo la concorrenza.



Devo avere un Gestore delle transazioni, che si occupa sia di gestire la concorrenza che di gestire l'affidabilità.

Il gestore del buffer per garantire affidabilità, deve fare in modo che quell'operazione venga **salvata in memoria di massa** (**è l'unico modo in cui garantisco effettivamente l'affidabilità**)

Ci saranno delle primitive che mi diranno "salvala nella base di dati" → salvala nella base di dati permanentemente.

Tecnologia delle basi di dati

- Gestione della memoria secondaria e del buffer
- Organizzazione fisica dei dati
- Gestione ("ottimizzazione") delle interrogazioni
- Controllo dell'affidabilità
- Controllo della concorrenza

Memoria principale e secondaria

Diamo per scontato di usarle entrambe: il DBMS è un prodotto software che mi permette di gestire i dati che ho in memoria secondaria, che non posso tenere tutti sulla memoria centrale (non devono essere volatili). Nel momento in cui uso il sistema, questo fa riferimento ai dati che ha in memoria centrale: i dati in memoria secondaria sono usabili solo se vengono trasferiti nella memoria centrale, tramite il buffer e il gestore dello stesso. Il mio programma, applicativo, usa la memoria centrale.

Ci deve essere l'opportuna gestione della memoria centrale e della memoria secondaria.

- Grandezza
- Persistenza

Il gioco è sempre la necessità di avere persistenza e la necessità di utilizzarli, quindi spostandoli.

Devo calcolare che l'informazione va e viene tra le due memorie.

Ho il buffer nella memoria centrale: come mi interfaccio con la memoria secondaria?

Memoria secondaria

È organizzato da blocchi di memoria di lunghezza fissa (alcuni KB)

Usiamo Blocco e Pagina come sinonimi

Le uniche operazioni sui dispositivi solo la lettura e la scrittura di una pagina, cioè dei dati di un blocco (cioè una stringa di byte): facciamo accessi a blocchi (ora) che sono anche accessi a pagine.

- Quando parliamo di costo di un'operazione di accesso: parliamo del tempo che impieghiamo per effettuare quell'operazione.
- Se lo faccio in memoria centrale, ho un certo costo, altrimenti ho un costo più alto (devo sempre ricordarmi che devo spostare i dati, se questi non sono già nel buffer)

Tempo di accesso alla memoria secondaria:

- Tempo di posizionamento della testina (10.50 ms)
- Tempo di latenza (tempo che intercorre tra la richiesta di un'informazione e il tempo che questo passi sotto la testina: 5-10 ms)
- Tempo di trasferimento (1-2 ms)

In media non meno di 10 ms.

Mi interessa sapere che: accedere alla memoria secondaria costa più tempo che accedere alla memoria centrale, perché mi interessa? perché in parte ho dati in memoria centrale (buffer) e così facendo, ottimizzando quali dati ho, risparmio tempo.

Il buffer

Il buffer è l'area della memoria centrale che gestisce il DBMS, dove posso caricare una porzione dei dati della mia base di dati. Quando sto facendo delle transazioni le informazioni che vengono considerate sono quelle che sono nel buffer: se le informazioni non sono lì, vuol dire che il mio sistema di gestione dovrà andarle a cercare nella memoria secondaria.

Il buffer è organizzato in pagine (ipotizziamo che la dimensione della pagina sia pari a quella del blocco: in generale ha dimensioni pari o multiple, ma non essendo questo l'aspetto che ci interessa, riduciamo pagina e blocchi).

Tanto più il buffer è grande, tanto più sono efficienti: devo accedere meno volte e ho più cose in memoria centrale.

La gestione ottimale del buffer è fondamentale: se riesco a fare in modo di avere nel buffer quello che mi interessa sarò più efficiente.

Cosa vuol dire gestire il buffer in maniera vantaggiosa?

Cercare di usare dei meccanismi di caricamento delle informazioni nel buffer che massimizzino la probabilità di avere le informazioni che mi servono nel buffer.

- Ridurre il numero di accessi che effettuo alla memoria secondaria (sono più veloce)
 - In caso di lettura, non devo accedere alla memoria secondaria
 - In caso di scrittura, posso decidere di differire la scrittura fisica:
 - Nel momento in cui scrivo nel buffer, scrivo anche nella memoria di massa: **affidabilità**.
 - Differisco la scrittura fisica, garantendo più **velocità** che affidabilità.
- Le politiche di gestione del buffer somiglano a quelle di gestione della memoria centrale di un SO. Obbediscono a un principio di località, in base a cui i dati referenziati di recente hanno una maggiore probabilità di essere referenziati in futuro.

Cerco di usare dei meccanismi di caricamento di info nel buffer che massimizzano la probabilità di trovare info di interesse proprio nel buffer.

Le info in più: cosa sia e l'area di memoria, accedere a memoria secondaria costa di più che la gestione. Ridurre il numero di accessi alla memoria secondaria significa essere più veloce nelle interrogazioni.

Nel caso in cui scrivo e i dati sono già nel buffer, ci sono due possibilità:

- La scrittura fatta in info temporanea del buffer e poi
 - a. O vado a scrivere su memoria di massa
 - b. Differisco la scrittura fisica solo se questa è compatibile con la gestione della affidabilità. (per quanto detto sopra)

Obbediscono a principio di località in base a cui i dati referenziati di recente hanno una maggiore probabilità di essere referenziati in futuro.

Nel momento in cui non è vero pescò in memoria di massa.

Il buffer interagisce con gestore dei metodi di accesso: e poi si interfaccia con gestore della memoria secondaria.

Il buffer offre una interfaccia complessa ai moduli di livello più alto. Ha anche bisogni delle informazioni del prevedibile utilizzo delle pagine e ha anche bisogno di sapere se l'info che deve fare deve essere effettuata immediatamente o no.

La memorizzazione del bonifico non me la puoi fare tra un'ora perché se il sistema crasha non va bene: **scrivo e rendo subito disponibile**.

La prenotazione dell'aula può essere differita nel tempo non cambia nulla se passa tempo.

Ricapitolando

Buffer

- Direttore per ogni pagina per mantenere file fisico e numero di blocco del file
- Per ogni pagina del buffer dobbiamo avere due variabili di stato
 - Un contatore dice quanti programmi utilizzano la pagina (parte memorizzata che corrisponde a blocco di informazioni)
 - Un bit che indica se la pagina è sporca = che è stata modificata

Ho fatto operazione su quella pagina e quella operazione deve essere memorizzata in memoria di massa. Quel che mi dice se ho delle operazioni che ho fatto e non ancora rese definitive con la memorizzazione in memoria secondaria

Intuitivamente: il buffer manager gestore del buffer:

Riceve richieste di lettura e scrittura di pagine, le esegue accedendo alla memoria secondaria se serve quando info non sta più nel buffer e esegue le primitive:

FIX
UNFIX
SERDIRTY
FORCE

Il gestore del buffer usa queste 4 primitive

Le politiche buffer si basano:

- Principio località dei dati: alta probabilità di usare gli stessi dati
- Legge 80/20: 80% delle operazioni usa lo stesso 20% dei dati - nell'80% DEI dati non devo accedere a memoria secondaria poiché 80% delle operazioni lavorano su stessi 20% dei dati.

Operazioni per la gestione del buffer

Fix

Richiesta di accesso alla pagina: un puntatore ci dice quante richieste di accesso abbiamo alla pagina → deve essere incrementato di uno

- Ritorna il riferimento della pagina al buffer manager

Set Dirty

Modifica l'altro puntatore: nel caso in cui una certa operazione venga modificata, abbiamo la necessità di modificare il BIT di stato relativo

Finché ho le pagine nel buffer e mi limito a leggerle, non ho problemi di affidabilità.

Comunica al buffer manager che la pagina è stata modificata (quindi so che deve essere caricata nella memoria secondaria)

Unfix

La transazione ha concluso l'uso della pagina: non ho più necessità di accedervi. Come prima, in fase di accesso, andiamo a decrementare il contatore di accesso. Ho fatto quello che dovevo fare.

La transazione si sgancia dalla pagina.

Force

Trasferisce in modo sincrono una pagina in memoria secondaria. Su richiesta del gestore dell'affidabilità viene memorizzata in quel momento: subito. Sto forzando la memorizzazione in memoria di massa immediata: garantire l'affidabilità, devo garantire che non perdo i dati.

Esecuzione

Se la pagina è presente nel buffer viene restituito l'indirizzo (avviene abbastanza spesso per l'affidabilità) e va ad incrementare il contatore.

Se la pagina non è nel buffer, abbiamo necessità di pescare la pagina dalla memoria di massa e caricarla nel buffer, questo vuol dire che dobbiamo capire se nel buffer ci sono pagine libere utilizzabili oppure no.

- Cerco una pagina libera (il contatore di accessi è zero, nessuna transazione sta considerando i dati della pagina - prima di fare qualsiasi cosa devo verificare che il bit di stato non mi dica che è stata modificata, altrimenti devo prima fare il caricamento dei dati in memoria di massa, così da garantire la memorizzazione e l'affidabilità) → ho creato lo spazio per caricare la pagina che mi interessa.
- Se non ci sono pagine libere (tutti i contatori sono maggiori di zero), ho due opzioni:
 1. Seleziono una pagina nel buffer che sacrifico per caricare una nuova pagina
 2. L'operazione viene posta in attesa che si liberino pagine nel buffer.

Nel momento in cui mi accorgo che una pagina è stata modificata devo capire come modificarla nella memoria secondaria: non posso garantirlo nella memoria primaria.

- Uso FORCE
- Quando il gestore del buffer lo ritiene opportuno o necessario (vengono usati specifici algoritmi per questa cosa) → esempio: se ho una pagina che si sta modificando, ha più modifiche, quando una pagina non è più acceduta posso pensare di salvarla (**risparmio**)

File System e DBMS

Il file system è il componente che gestisce la memoria secondaria, quindi il DBMS si appoggia al file system, per caricare, salvare, e gestire i blocchi.

- Le funzioni che ne usiamo sono comunque limitate
- Le mie informazioni finiscono ovviamente in dei file: il modo in cui viene gestito il contenuto del file (come le salvo, come le organizzo, come le cancello) è sempre gestito dal DBMS.
- Il DBMS gestisce i blocchi dei file allocati come se fossero un unico grande spazio di memoria e costruisce lì le strutture fisiche con cui implementa le relazioni.
- Il DBMS ha necessità di creare file di grandi dimensioni per memorizzare parte o tutta la base di dati.

Dobbiamo immaginare i file organizzati in blocchi all'interno dei quali sono memorizzate le tuple:

Come?

- In unico file
- Blocchi contigui con tuple della stessa relazione
- Ecc.

Le tuple della base di dati finiscono memorizzati nei file, questo è quello che ci interessa.

Il DBMS quindi deve organizzare le tuple nei file. Ci sono diversi modi di farlo, ma con aspetti comuni che valgono in qualsiasi tipo di organizzazione:

- Ciascuna pagina, quindi ciascun blocco, ha una parte di **informazione utile** (i dati veri e propri, le tuple) e una serie di **informazioni di controllo** (quindi ciò che il DBMS usa per gestire i dati veri e propri e come farlo)



La tupla è la singola riga della tabella, possiamo chiamarla anche record (insieme di informazioni strutturate in un certo modo)

I blocchi e i record hanno in generale dimensioni diverse, che possono variare: non posso pensare che corrispondano, anche perché il blocco dipende dal file system, la dimensione del record invece dipende dall'applicazione.

Fattore di blocco

È il numero di record che stanno in un blocco.

La prima approssimazione è pari al rapporto tra la dimensione del blocco e quella del record (media):

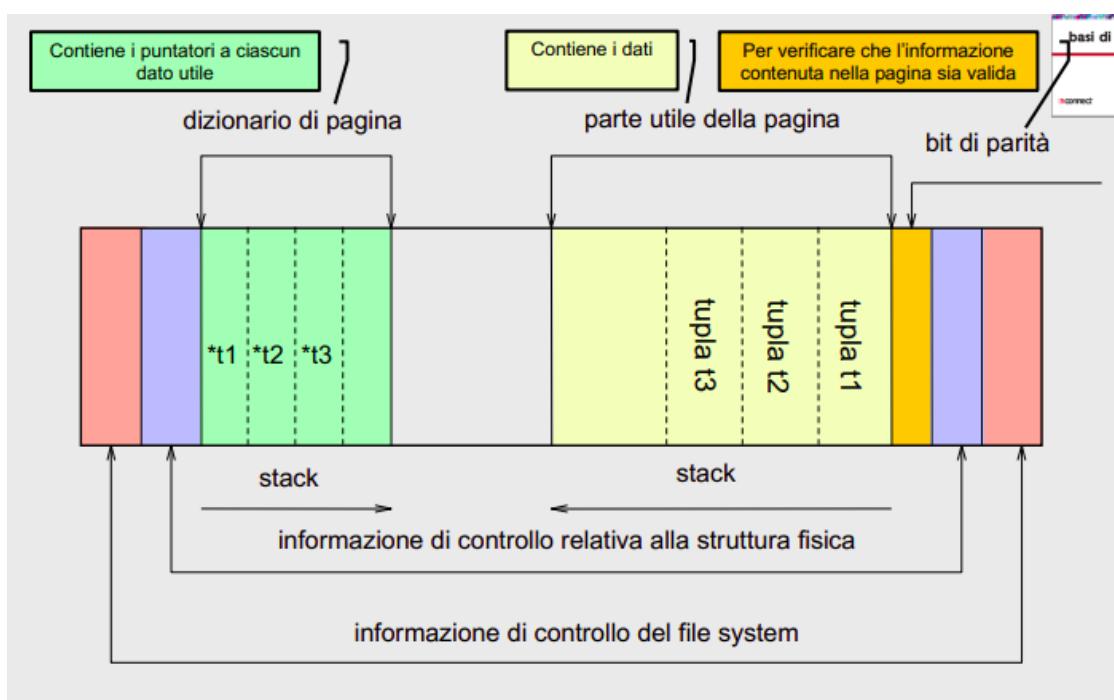
- L_B : dimensione del blocco
- L_R : dimensione media di un record (nel file prendiamo una lunghezza fissa)

Prendo la parte intera, arrotondata per difetto.

Rimangono degli spazi residui che in alcuni casi possono essere utilizzati.

Per l'organizzazione delle tuple nella pagina abbiamo varie possibilità, legate ai metodi di accesso, a come il sistema lo gestisce.

Esempio:



Il tipo di movimento è verso l'interno, lo spazio grigio al centro è quello libero.

Coincide ovviamente con un blocco di memoria gestito dal file system:

- ha un header e un trailer, che consistono in informazioni di controllo del file system

Alllo stesso modo, una volta creato, il file viene gestito dal DBMS, quindi il contenuto del blocco contiene anche l'informazione di controllo relativa alla struttura fisica:

- header e trailer, sempre relativi al blocco, con informazioni utili al DBMS

Quali sono le strutture che il DBMS può utilizzare per organizzare i dati nel blocco?

Stabilisco il criterio secondo cui le tuple sono disposte nel file.

Tre principali categorie di strutture:

- Sequenziali
- Ad accesso calcolato (tavole hash)
 - Disposizione in posizioni che dipendono dall'esecuzione di un algoritmo
- Ad albero
 - Usate come strutture secondarie

Strutture sequenziali

Nella struttura sequenziale l'ordinamento delle tuple può essere:

- **Seriale**
Ordinamento fisico ma non logico in base all'immissione
- **Array**
- **Ordinata**

Struttura seriale (disordinata)

È una struttura semplice, ma anche la più diffusa perché in fase di inserimento non mi crea problemi, le tuple vengono inserite nell'ordine in cui arrivano.

Chiamata anche:

- Entry Sequences
- File Heap
- File Disordinato

Gli inserimenti vengono effettuati:

- in coda
- al posto di record cancellati

È molto diffusa nelle basi di dati ma associata a indici secondari. Cerco di mantenere le cose più efficienti possibili.

Perché?

Le ricerche sono poco efficienti, richiedono una scansione sequenziale di tutti i record. Per questo sono spesso usate insieme a una struttura secondaria di accesso che mi permette di essere più efficiente.

È molto efficiente però per le azioni di inserimento.

Ad Array

Possibili solo se le tuple hanno lunghezza fissa, perché alloco spazi della mia memoria a cui associo un indice. Ogni blocco ha un numero fisso di posizioni disponibili, ogni tupla ha un indice e viene posta nell'indice associato.

Sembra ordinato, è vantaggioso dal punto di vista del recupero dell'informazione, ma dal punto di vista della memoria non è efficiente. Quindi non viene quasi mai utilizzato.

Strutture ordinarie

La memorizzazione dei record avviene secondo un ordinamento fisico coerente con quello logico: la chiave, il cognome, la matricola, ecc.

Permettono ricerche binarie ma solo fino a un certo punto, perché non abbiamo indici. Si usano solo in combinazione con questi ultimi infatti.

Strutture ad accesso calcolato

Parliamo di file Hash, abbiamo l'allocazione delle tuple in posizioni che dipendono dall'algoritmo.

Sono vantaggiose perché permettono un accesso molto efficiente.



Obiettivo: accesso diretto ad un insieme di record sulla base del valore di un campo (detto chiave, che per semplicità supponiamo identificante, ma non è necessario)

Se i possibili valori della chiave sono in numero paragonabile al numero di record (e corrispondono ad un "tipo indice") allora usiamo un array; ad esempio: università con 1000 studenti e numeri di matricola compresi fra 1 e 1000 o poco più e file con tutti gli studenti.

Se i possibili valori della chiave sono molti di più di quelli effettivamente utilizzati, non possiamo usare l'array (spreco).

Cosa facciamo?

La **funzione di hash** usa un mapping della chiave sulla posizione, senza sprecare spazio, trasformo i valori della chiave in indici dell'array.

- Associa ad ogni valore della chiave un "indirizzo", in uno spazio di dimensione paragonabile (leggermente superiore) rispetto a quello strettamente necessario
- Poiché il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi ("lo spazio delle chiavi è più grande dello spazio degli indirizzi"), la funzione non può essere iniettiva e quindi esiste la possibilità di collisioni (chiavi diverse che corrispondono allo stesso indirizzo) - mi trovo con una funzione di hash che per due chiavi che sono diverse, finiscono nello stesso posto (lo posso gestire)
- Le buone funzioni hash distribuiscono in modo causale e uniforme, riducendo le probabilità di collisione (che si riduce aumentando lo spazio ridondante): la tavola ideale non ha collisioni, ma non la posso avere perché sarebbe iniettiva.

La logica è: avendo le tuple, considero un certo valore, per capire dove memorizzarle nel file, uso una funzione di hash: questa mi dirà, dato un valore, dove l'ho memorizzato (l'indice)

Come gestisco le collisioni?

- La prima finisce effettivamente nel posto
- La seconda la gestisco: salto di uno, trovo il primo posto vuoto, trovo altri modi.

Ho varie tecniche:

- Posizioni successive disponibili
- Una tabella di overflow
- Funzioni di hash alternative

Nota:

- Ho necessità di definire le collisioni (ci sono quasi sempre!)
- Potrei anche essere fortunata e non avere collisioni.
- Il numero di collisioni, decresce rispetto alla molteplicità delle collisioni:
 - Molteplicità vuol dire quante chiavi diverse mi portano nella stessa posizione
 - Il numero di collisioni decresce quando questo aumenta, perché la probabilità che io abbia una molteplicità alta, è bassa - altrimenti la mia funzione di hash non è buona.

File Hash

Numero di accessi

Per calcolare il numero medio di accessi devo fare il numero di accessi / numero dei record.

Nell'organizzazione di file hash vengono applicate le stesse caratteristiche della funzione hash, l'idea è la stessa ma dobbiamo pensare che l'organizzazione è a blocchi → la funzione ci tornerà il numero di blocchi in cui dobbiamo sistemare la tupla. In questo caso il punto in cui memorizzare il record è uno dei blocchi del file.

Tutti i ragionamenti sono quindi analoghi:

- Il file è suddiviso in blocchi (per noi corrisponde a una pagina di memoria)
- I blocchi sono riempiti solo in parte, non è detto che in un blocco vengano inseriti tutti:

Fattore di riempimento è la frazione dello spazio fisico disponibile:

quanto riempio le posizioni che ho a disposizione (numero di tuple necessarie / numero di posti)

- Il fattore di blocco è il rapporto tra la dimensione di un blocco e la dimensione del record:

Risponde alla domanda quanti record possono stare nel mio blocco

- T = numero di tuple
- FbB = fattore di blocco
- FdR = fattore di riempimento

Posso calcolare il numero di blocchi da cui è composto il file:

$$B = \lceil \frac{T}{FdR \cdot FdB} \rceil$$

La mia funzione di hash mi tornerà un numero compreso tra 0 e $B - 1$ cioè data la chiave (es. matricola) mi dirà in quale dei blocchi la tupla andrà a finire.

Se ho una collisione?

Semplicemente vado ad allocare nei record del blocco fino al riempimento dello spazio disponibile.



È un'allocazione sui blocchi!

Quando il blocco viene esaurito ho il primo problema. Cosa faccio?

Allocò un nuovo blocco allegato al precedente: non posso più metterle nello stesso blocco e vado ad allocarle in uno successivo, viene gestito dalle informazioni di controllo relative alla struttura fisica, gestite dal DBMS. Tengo traccia dei collegamenti tra blocchi.

Questo meccanismo si chiama **Catene di Overflow**.

Il file hash è l'organizzazione più efficiente per l'accesso diretto (puntuale: prendo una tupla e vedo dov'è) ho un costo superiore all'unitario quindi è molto efficiente.

Questo tipo di organizzazione invece non è efficiente per accessi collegati ad intervalli.

Organizzazione ad albero

Può essere utilizzata per realizzare due tipi di strutture:

- Strutture primarie (contenenti i dati) - leggermente meno efficiente dell'hash
- Strutture secondarie (favorisce l'accesso ai dati tramite indici)

Mantiene l'accesso simil-puntuale anche per gli intervalli.

Indici



Definizione:

Un indice è una struttura ausiliaria che ci permette di accedere in modo facilitato ad un'informazione.

L'idea è di permettere un accesso facilitato alle informazioni, se parliamo di accesso ad un file avremo una coppia (*key, position*).

Un indice di un file è un altro file organizzato con record di due campi: la chiave e l'indirizzo del record o del blocco relativo al record.

Tipi di indici

- **Indice primario**

Contiene al suo interno i dati. Ci dice già com'è organizzato un blocco, perché ci dà l'informazione diretta sul dato.

- **Indice secondario**

Definito su un campo di ordinamento diverso da quello di ordinazione

- **Indice denso**

Contiene un record per ciascun valore del campo chiave

- **Indice sparso**

Contiene un numero di record inferiore ai valori effettivi della chiave, esistono valori della chiave che non sono presenti nell'indice.

Un indice secondario deve essere denso perché non è detto che valori consecutivi della chiave siano consecutivi, cosa che succede invece per il primario.

Ogni file può avere al più un indice primario e un numero qualunque di indici secondari.

Prestazioni

- L'accesso diretto sulla chiave è efficiente
- Scansione sequenziale ordinata è efficiente
- Non è efficiente la modifica della chiave, gli inserimenti, le eliminazioni

Indici Multi livello

Sono organizzati a più livelli, organizzati con la logica identica a prima, ma su più livelli.

Tutte le strutture di cui abbiamo parlato sono organizzate ma poco flessibili quando c'è molta dinamicità - è naturale per me pensare di aggiungere, cancellare e modificare.

Gli indici che i DBMS utilizzano sono più sofisticati: usano strutture ad albero dinamiche, efficienti in materia di aggiornamenti.

Strutture ad albero dinamiche

Ha un nodo radice, nodi intermedi e nodi foglia.

Ogni nodo coincide con una pagina o blocco del file, ogni nodo ha un numero di figli che dipende dall'ampiezza della pagina.

Il numero di livelli è limitato e il collegamento avviene tramite puntatori.

Il requisito importante: **BILANCIATO**



La lunghezza del cammino che collega una radice a una foglia è costante (per raggiungere qualsiasi foglia, il mio cammino ha sempre la stessa lunghezza)

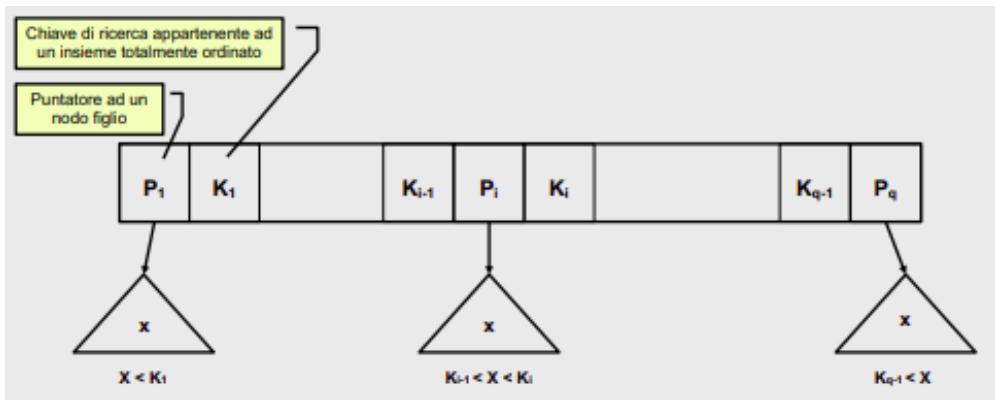
- Alberi di ricerca bilanciati (B-Tree)
 - Alberi binari di ricerca
 - Alberi n-ari di ricerca
 - Alberi n-ari di ricerca bilanciati

Albero binario di ricerca

Albero etichettato in modo che per ogni nodo dell'albero, sappiamo che il sotto-albero di sinistra contiene solo etichette minori del nodo e il sotto-albero di destra solo etichette maggiori del nodo.

Albero di ricerca di ordine p

Quando passo da ordine 2 (albero binario) a ordine maggiore di 2 (esempio p), posso avere fino a p figli (e ha fino a $p - 1$ etichette ordinate)



Nell'i-esimo sotto-albero abbiamo tutte etichette maggiori della (i-1)- esima etichetta e minori della i-esima.

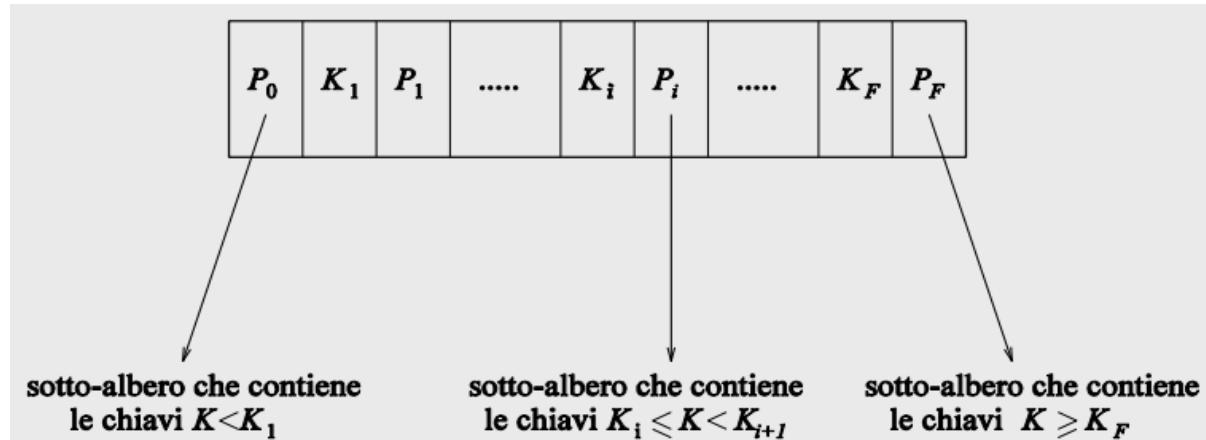
Ogni ricerca come prima comporta la visita di un cammino dalla radice alla foglia, il B-Tree è un albero di ricerca fatto come quello appena visto, ma bilanciato, ovvero ha un riempimento parziale (intorno al 70 %) e in caso di sbilanciamento viene ri-bilanciato.

Albero bilanciato

Il numero di puntatori $F + 1$ si chiama Fan Out.

F valori ordinati di chiave

- Ogni chiave $K_i (1 \leq i \leq F)$ è seguita da un puntatore P_i
- K_1 è preceduta da un puntatore P_0



Meccanismo di ricerca

Seguo i puntatori dalla radice (come nell'albero binario, ma ne ho di più). Se ho una chiave di ricerca vado a vedere

Se la chiave di ricerca è V , ad ogni nodo intermedio

- Se $V < K_1 \rightarrow$ si segue il puntatore P_0
- Se $V \geq K_F \rightarrow$ si segue il puntatore P_F
- Altrimenti si segue il puntatore P_j tale che $K_j \leq V < K_{j+1}$

La ricerca procede fino ai nodi foglia, se è indice primario, conterrà i dati, se è secondario, i nodi foglia contengono i puntatori ai blocchi che contengono le mie tuple.

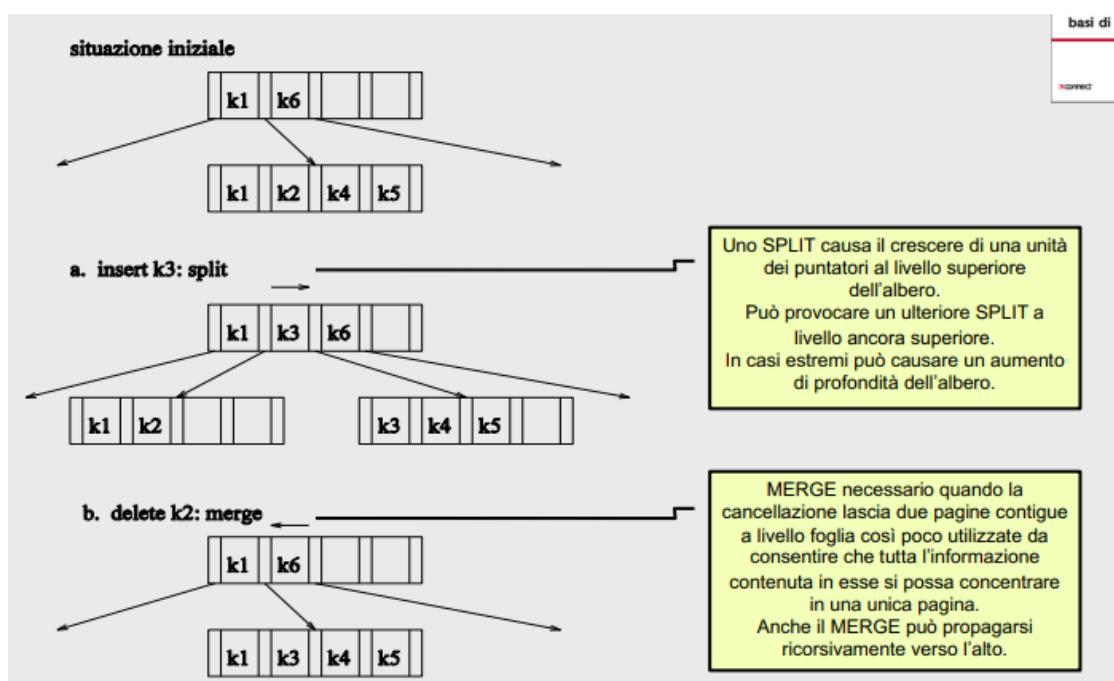
Se devo inserire o cancellare qualcosa, devo arrivare alla foglia e capire cosa posso fare.

Inserimento:

- C'è posto nella foglia?
 - Sì, inserisco
 - No, vado al nodo suddiviso SPLIT
 - Necessito di un nodo in più per il nodo genitore
 - Se non c'è ancora posto, divido ancora

Cancellazione:

- Potrei trovarmi ad avere situazioni sbilanciate: MERGE (faccio merge tra nodi, li unisco e verifico → riduco il numero di puntatori e organizzo l'albero)



Due versioni

- B-tree

I nodi intermedi possono avere puntatori direttamente ai dati e i nodi foglia non sono legati sequenzialmente

- B+tree

I nodi foglia sono collegati in una lista, che quindi è ordinata tramite l'ordine imposto dalla chiave.

Questo li rende vantaggiosi per le ricerche per intervallo, una volta che ho fatto la discesa di ricerca sull'albero, mi basterà scorrere per aver fatto la ricerca sull'intervallo. Queste sono utili e vengono fatte abbastanza spesso.

Domande esame

- Cos'è un DBMS ecc?
- Quali sono le strutture organizzative di un file? (Heap, Alberi, ecc.)

- Indicare i tipi di indice
- Esercizi su Tree e B-tree

3 domande + 3 esercizi

1 sui B-Tree

Gestore delle interrogazioni

Modulo cruciale del DBMS: quando scriviamo la query SQL cosa succede?

Una volta organizzati i dati a livello fisico, ci viene permesso di eseguire in maniera efficiente interrogazioni specificate a livello molto alto: noi scriviamo in maniera dichiarativa le caratteristiche che i dati devono avere per avere un risultato.

Noi non ci preoccupiamo di come i dati sono stati memorizzati, specifichiamo cosa stiamo cercando e ci aspettiamo di ottenere un insieme di tuple.

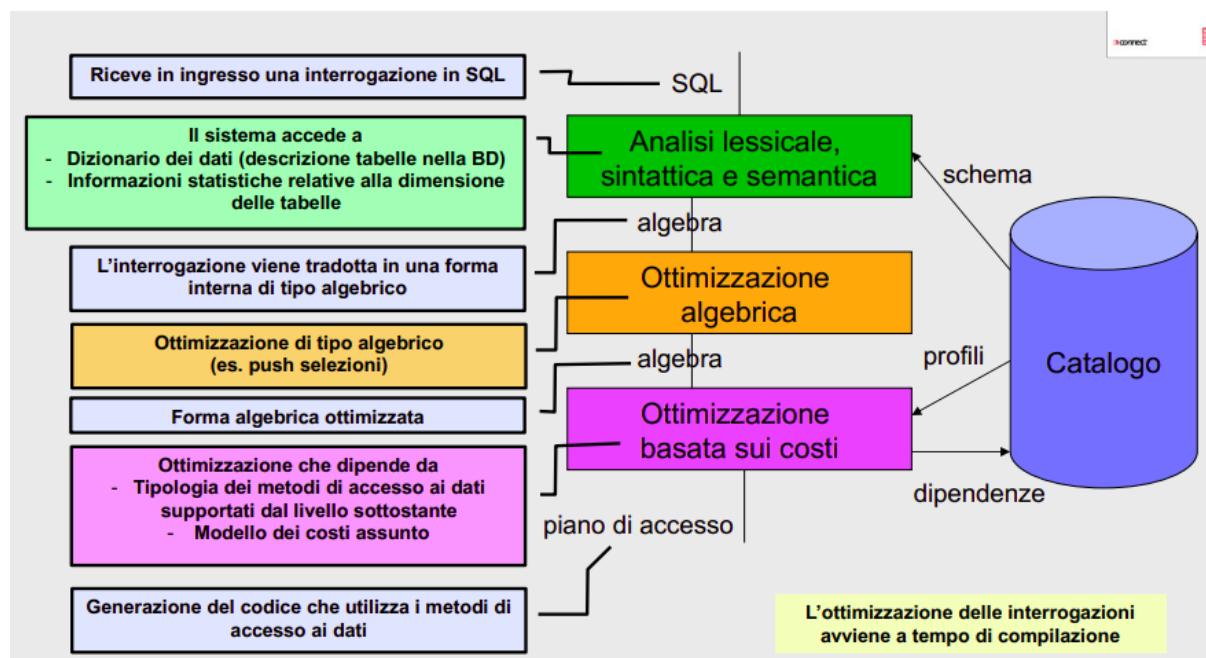
Abbiamo già parlato dell'ottimizzazione delle interrogazioni, quella parte che non può essere specificata facendo SQL, viene eseguita in forma algebrica - un linguaggio procedurale in cui decido in che modo vengono eseguite le queries.

IL DBMS passa da una rappresentazione dichiarativa ad una algebrica:

Mette a confronto varie possibili scelte di esecuzione e poi sceglie la più ottimizzata.

Esecuzione delle interrogazioni

I passaggi avvengono a tempo di compilazione:



Input: Query SQL, Output: Piano di accesso ai dati per come sono stati memorizzati

1. Analisi lessicale, sintattica e semantica

Il sistema accede al dizionario dei dati - il sistema saprà con che oggetti avrà a che fare: il risultato è una **rappresentazione della query SQL in un formato interno di tipo algebrico**.

Passiamo a vedere come il mio sistema può trovare quello che voglio.

2. Ottimizzazione algebrica

Il risultato è una forma algebrica ottimizzata, la query algebrica (che corrisponde alla query SQL) con la forma più efficiente possibile.

3. Ottimizzazione basata sui costi

Questo tipo di ottimizzazione dipende molto dall'organizzazione fisica dei nostri dati, da come sono disposti, dal modello assunto, ecc. I costi che dobbiamo supportare sono il tempo impiegato e le operazioni necessarie ad ottenere quello che voglio.

Dipende da:

- Strutture di memorizzazione
- Modello dei costi prescelto

Le informazioni che il DBMS ha a disposizione si chiamano **profili delle relazioni** (ci danno tutte le informazioni quantitative relative ad una tabella) sono memorizzate nel dizionario della base di dati (es. numero di tuple, dimensione delle tuple, ecc.). Queste sono periodicamente aggiornate, perché i dati cambiano continuamente. In particolare questo profilo si chiama **Catalogo**.

Infine quindi avremo generato il codice che ottiene le informazioni che mi servono.

Ottimizzazione algebrica

Si basa su una nozione di equivalenza: cerco di scrivere in algebra in un formato equivalente a quello di partenza ma più efficiente.

Il DBMS cerca di costruire espressioni algebriche equivalenti alla query SQL richiesta, ma che siano meno costose.



Idea alla base:

Ridurre il più possibile i risultati intermedi → mi porto dietro solo quello che mi serve.

- Fare il prima possibile le selezioni - push selection down
- Fare il prima possibile le proiezioni - push projection down

Voglio ridurre al minimo le cardinalità dei risultati intermedi (es. prima la selezione del join)

Come sono rappresentate le nostre operazioni interne?

Abbiamo degli alberi:

- Foglie sono i dati
- I nodi intermedi sono le operazioni
- La radice è l'ultimo operatore per ottenere il risultato

Rispecchia la rappresentazione usata.

Tecniche e procedure

- Decomporre le soluzioni congiuntive in selezioni atomiche (anticipare le selezioni il più possibile)

- Se ho una sequenza di selezioni, anticipare le più selettive
- Anticipare le proiezioni il più possibile
- Combinare prodotto cartesiano e selezioni nel join

Interpretazione - prima mappo in questo modo sequenziale, **poi** ottimizzo:

- prodotto cartesiano (FROM)
- selezione (WHERE)
- proiezione (SELECT)

Nota: Il join è sempre preferibile al prodotto cartesiano completo

- Vengono anticipate le selezioni più selettive (esempio di selezione di $H = 7$, prima di $I > 2$), in cascata ho del guadagno da questa cosa
- Anticipo anche le proiezioni, così da non portarmi dietro attributi che non mi servono. Quelli che mi servono sono quelli che saranno successivamente coinvolti sui join o fanno parte del risultato.

La dimensione del risultato intermedio di ogni passaggio quindi è nettamente inferiore a quello dei risultati intermedi non ottimizzati.

Se pensiamo a tabelle molto grandi passare da un milione a cento mila tuple, fa tantissima differenza.

Esecuzione delle operazioni

I DBMS implementano gli operatori dell'algebra relazionale (o meglio, loro combinazioni) per mezzo di operazioni di livello abbastanza basso, che però possono implementare vari operatori "in un colpo solo"

- Operatori fondamentali:
 - scansione
 - accesso diretto
- A livello più alto:
 - ordinamento
- Ancora più alto
 - join

Scansione

Utilizzata per la proiezione o per la selezione

Accesso diretto

Quando posso accedere direttamente a una tupla (esempio accesso alla chiave, ecc.): non mi serve andare a scandire in maniera sequenziale tutti i record, perché so esattamente cosa sto cercando e perché le strutture fisiche me lo permettono (indici, strutture hash me lo permettono - un file disordinato, sicuramente no)

Il tipo di operazione quindi dipende fortemente da come ho definito la struttura dati e anche l'ottimizzazione ne dipende.

Ordinamento

Quando mi può essere utile? Se il risultato deve essere ordinato (es. ordine alfabetico)

Devo poter accedere e ordinare le informazioni.

Altri casi possono essere avere ad esempio la proiezione con l'eliminazione dei duplicati - solo dall'ordine me ne accordo.

Il DBMS utilizza **varianti degli algoritmi di ordinamento classici**, tenendo conto delle caratteristiche della memoria secondaria e dell'albero.

Join

È l'operazione più costosa perché il numero di tuple coinvolte può essere molto alto, ad esempio join su campi non chiave - il numero intermedio di tuple può essere paragonabile al prodotto cartesiano.

Abbiamo diversi algoritmi per realizzare il join:

- *nested-loop*
- *merge-scan*
- *hash-based*

Sfrutta la caratteristica di essere basato sui valori, quindi la realizzazione del join è importantissima.

Nesting Loop



Propone una scansione nidificata.

Una tabella viene definita come esterna e una come interna, l'algoritmo esegue una scansione della tabella esterna e per ogni tupla prende il valore dell'attributo di join e va a vedere nella tabella interna dov'è presente quel valore.

Se nella tabella interna ho una struttura hash o ho un indice sull'attributo, avrò sicuramente il lavoro facilitato, se invece non ne ho devo fare la scansione dell'intera tabella.

Avere o non avere un indice dipende da chi ha progettato la base di dati: lo definisco se ipotizzo che le scansioni su quell'attributo siano molto frequenti.

In fase di progettazione, in base alle interrogazioni che presumo verranno fatte, vado a costituire un indice (anche successivamente)

Merge Scan



Abbiamo una scansione parallela.

Man mano mettiamo insieme le tuple che hanno lo stesso valore sull'attributo di join: particolarmente efficiente quando le tabelle sono già ordinate o sono definiti gli indici adeguati.

Metto insieme ovviamente le tuple con lo stesso valore sull'attributo di join e le porto sul risultato. La scansione avviene parallelamente e contemporaneamente, secondo l'indicazione del join.

Hash Join



Si basa sull'idea di avere a disposizione una funzione di hash.

Agli attributi di ogni tabella necessari al join viene applicata la funzione di hash, così che venga corrisposto un certo risultato ad una partizione delle tabelle, le tuple che hanno gli stessi valori di attributo, finiscono nella stessa partizione con lo stesso indice. Così posso combinare direttamente le partizioni con lo stesso indice.

Pre-elaborazione: applico la funzione hash, ho quindi un'elaborazione della tabella → l'unico scopo è creare delle partizioni delle tuple delle nostre tabelle, così che siano collocate nella stessa partizione.

Ciascun algoritmo ha dei **costi diversi, e dipendono dal tipo di struttura** al di sotto:

- Ho la presenza di indici? (es. Nested Loop ha due costi molto diversi in questo caso)

Ottimizzazione basata sui costi

In questa fase il DBMS deve tener conto di varie dimensioni di ottimizzazione, che dipendono dal tipo di scelta fatta in costruzione.

- Operazioni da eseguire
 - Dobbiamo fare una scansione?
 - Possiamo fare un accesso diretto?
- In che ordine è opportuno fare join in cascata
 - Ipotizzando di avere più algoritmi di join a disposizione, posso pensare di usarne uno piuttosto che un altro
- Il tipo di architettura può anche influenzare

Il DBMS baserà la scelta in funzione di una serie di **formule di costo approssimate**.

Processo di ottimizzazione

Per decidere qual è il **piano di esecuzione** migliore, a fronte di una determinata interrogazione che può essere eseguita con piano di accesso diverso.

Costruisco un albero di decisione con tutte le alternative (**piani di esecuzione**): **Albero delle Alternative**

- Ogni nodo ha una particolare opzione, che può essere combinata con altre opzioni
Le dimensioni dell'albero crescono in base alle opzioni presenti
- In ogni nodo foglia, che corrisponde a un certo percorso di scelte, avremo una specifica strategia di esecuzione di una certa interrogazione.
- Nelle foglie avremo anche il costo relativo a quel piano di esecuzione, andiamo a scegliere il nodo foglia con il costo minore tra tutti e seguiremo il percorso che ci arriva.

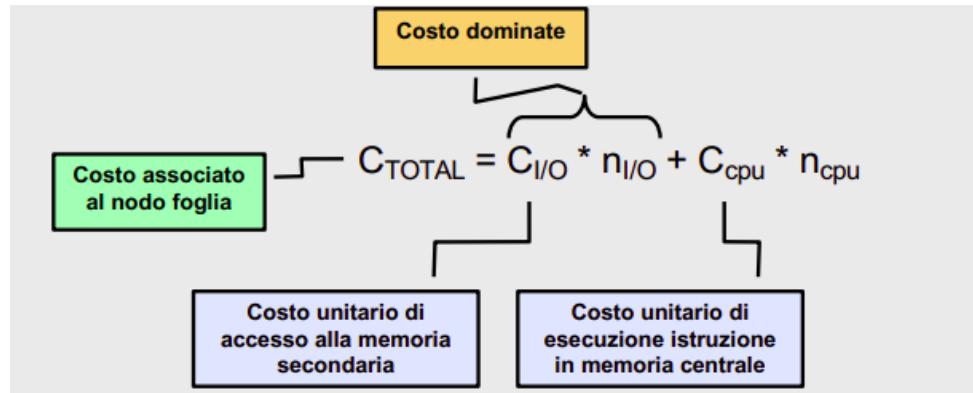
Una volta che ho costruito il mio albero devo valutare il piano che costa meno. Di solito l'ottimizzatore trova una buona soluzione, non quella ottima:

vengono considerate buone soluzioni quelle che rientrano nello stesso ordine di grandezza della soluzione ottimale.

Escludo tutti i sotto alberi che hanno un costo peggiore del costo di una strategia globale già individuata.

Le formule di costo associano a ciascuna operazione intermedia un costo in termini di

- Operazioni di ingresso/uscita
- Istruzioni necessarie per valutare il risultato dell'interrogazione



Tutte sono stime: non posso sapere a priori quante volte ho accesso alla memoria secondaria, ma posso fare una stima su quello che sarà il costo finale.

Progettazione fisica

Decido in che modo le mie tabelle vengono mappate sulle strutture fisiche.

Input: lo schema logico e informazioni sul carico applicativo

Output: schema fisico, costituito dalle definizione delle relazioni con le relative strutture fisiche (e molti parametri, spesso legati allo specifico DBMS)

Questi parametri sono diversi da DBMS a DBMS, ma ci sono una serie di cose comuni a tutti:

- Possibilità di avere degli indici
 - Di solito le chiavi primarie sono coinvolte nella definizione di indici, perché sono coinvolte sia nelle operazioni di selezione che in quelle di join
 - DBMS prevedono o suggeriscono di farlo
- Nel momento in cui mi trovo a definire un indice, so che quell'indice può essere utile in query ricorrenti, quindi all'inizio devo fare un'analisi, pensando a quali query possono essere più ricorrenti. Devo informarmi sulle operazioni che saranno fatte su quei dati (mantenere o eliminare ridondanze, capire i domini da attribuire agli attributi, ecc.)

`show plan` : comando SQL che permette di vedere gli indici.

FINE PAGINA

B-tree - recap

Non abbiamo chiavi duplicate tra i vari livelli dell'albero.

- Nei B-tree i nodi intermedi possono puntare direttamente ai dati
- Per ogni chiave k_i nel nodo abbiamo anche il puntatore direttamente al record

La definizione è un certo puntatore p_i punterà ad un certo albero in cui le chiavi sono comprese strettamente tra k_j e $k_j + 1$: non serve duplicare l'informazione.

La radice in un nodo di ordine m (con m = 3 altrimenti sarebbe binario) - m è Fan-Out

ha un numero di nodi figli compresi tra 0 e m, e un numero di chiavi compresi tra 0 e m -1

Ho sempre un puntatore in più

Se definisco rispetto alle chiavi, queste sono n e i puntatori sono n + 1

Ogni nodo interno ha un numero di figli compreso tra la metà (arrotondata all'intero superiore) e m

Tutte le foglie si trovano nello stesso livello

Come inserisco una chiave?

Vuol dire che sto inserendo una tupla

1. Se la chiave è presente si verifica, altrimenti devo scendere alla figlia appropriata.
2. Se il nodo non è completo semplicemente aggiungo la chiave
3. Se il nodo è completo opero lo **splitting dei nodi**
 - a. Si considera il valore mediano tra le chiavi del nodo completo più la chiave da inserire (o che è in posto mediano) e sarà il valore che inserisco nel nodo padre
 - b. Se il padre è completo si esegue di nuovo lo splitting (ricorsivo)

Cancellazione di una chiave

Vuol dire che sto cancellando una tupla

Vado a cercare dov'è la chiave da cancellare:

1. Se è in un nodo foglia allora cancello la foglia e vado a controllare (al passo 3) il bilanciamento dell'albero
2. Se non è foglia, la devo sostituire con o la chiave più piccola tra gli elementi maggiori o la chiave più grande tra gli elementi minori

Come scelgo? È una mia scelta.

3. Controllo che l'albero sia bilanciato: devo guardare ciò che ho ottenuto, e spostare le chiavi in modo da avere tutto bilanciato.

Esercizio sui B-tree

Dato il B-tree in figura con Fan-Out pari a 4, eseguire le perazioni indicate mantenendo l'albero bilanciato.

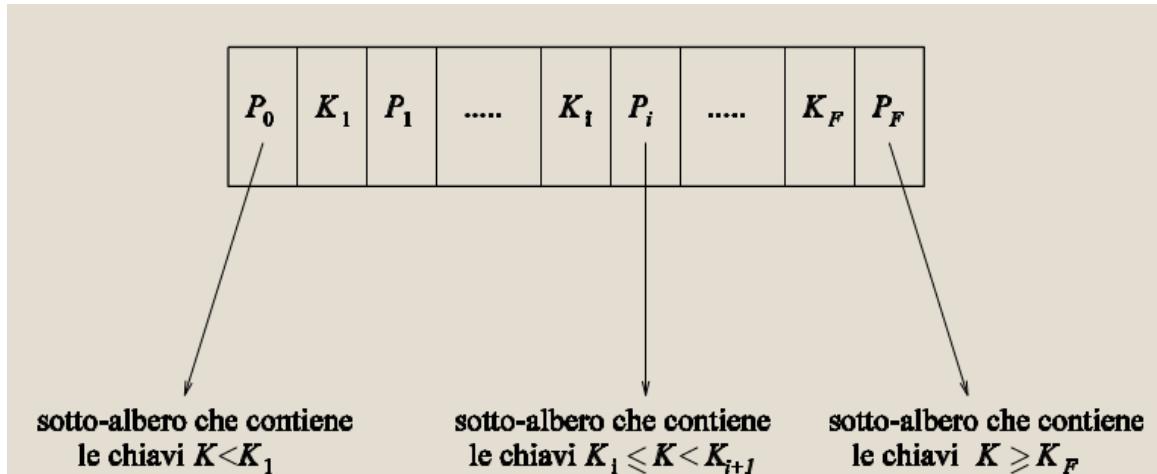
Non devi fare la cosa più pulita ma quella che mi permette di fare meno azioni successivamente:

- esempio: è inutile ribilanciare l'albero (ottenendone uno completo) se poi per aggiungere una tupla dovrò comunque tornare a quello che avevo prima

B+tree recap

B+ Tree:

- Le foglie sono collegate in una lista
- I puntatori ai dati sono solo nelle foglie
- L'inserimento delle chiavi avviene nei nodi foglia
- Ottimi per le ricerche su intervalli
- Molto usati nei DBMS



Inserimento

Non provoca problemi quando la nuova chiave si può inserire in una foglia non satura

Quando la foglia non ha spazio disponibile, si esegue lo SPLIT della foglia

- Si suddivide l'informazione presente nella foglia e la nuova informazione in due (in modo equilibrato) allocando due foglie al posto di una
- Il valore mediano delle chiavi si inserisce anche (in copia) nel padre
- SPLIT causa il crescere del numero dei puntatori al livello superiore
- Lo SPLIT può dover essere ripetuto sul nodo padre
- Lo SPLIT del padre (nodo non foglia) viene gestito come per i BTree
 - Si suddivide l'informazione in due, mettendo le chiavi maggiori del valore mediano nel nuovo nodo
 - Il valore mediano delle chiavi viene spostato nel padre

Rimozione di una chiave

Se la rimozione della chiave lascia due nodi foglia con spazio inutilizzato da consentire che tutta l'informazione in esse presente possa essere concentrata in una sola foglia

- Operazione di MERGE
 - Richiede la modifica della disposizione dei puntatori
 - MERGE causa il decrescere del numero dei puntatori al livello superiore
 - MERGE può dover essere ripetuto sul nodo padre

Bilanciamento

Se le modifiche causano differenze tra le lunghezze dei cammini tra radice e foglie

Opportuno ribilanciare l'albero

Gestione delle transazioni

In particolare risulta utile per garantire quelle caratteristiche che il DBMS ci mette a disposizione rispetto al fatto che ciò che viene memorizzato sia consistente nel tempo, dalla presenza di malfunzionamenti e accessi concorrenti.



*Abbiamo garanzia di **consistenza***

Cos'è una transazione?

È un'unità di lavoro elementare che viene gestita da un'applicazione.

Posso ovviamente svolgere più transazioni in contemporanea.

Questa unità elementare garantisce:

- Correttezza
- Robustezza
- Isolamento

Struttura

Una transazione ha un **Inizio**: `begin-transaction` (`start transaction` in SQL)

Ha una **Fine**: `end-transaction` (non esplicita in SQL)

Garantisce che le operazioni all'interno vengono svolte tutte insieme, e mi garantisce che o vengono portate a termine tutte, oppure nessuna.

La transazione termina correttamente quando viene eseguita l'operazione `commit work`

Invece l'aborto della transazione avviene tramite il comando `rollback work` : disfo tutto quello che ho fatto.

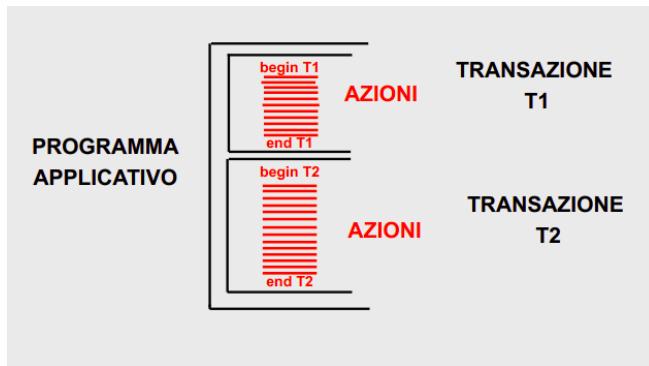
Sistema Transazionale

Differenza tra applicazione e transazione

Un sistema che usa questo tipo di struttura si chiama

- Un sistema transazionale (OLTP) è in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti

Different from OLTP = On-Line Transaction Process which instead concerns large amounts of



dati online.

Esempio

```
start transaction;
update ContoCorrente
  set Saldo = Saldo + 10
  where NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo - 1
  where NumConto = 42177;
commit work;
```

Esempio più complesso:

```
start transaction;
update ContoCorrente
  set Saldo = Saldo + 10
  where NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo - 10
  where NumConto = 42177;
select Saldo into A
  from ContoCorrente
  where NumConto = 42177;
if (A>=0)  then commit work
else rollback work;
```

Transazione SQL più articolata che prevede anche strutture di controllo. Può verificarsi il caso in cui, dopo aver effettuato alcune operazioni, il condizioni verificate portano alla decisione che gli aggiornamenti devono essere annullati. ROLLBACK

Se Saldo negativo

Nel momento in cui dico `rollback` allora la transazione non è andata a buon fine e lo stato della base di dati dopo deve essere uguale allo stato prima del mio `start transaction`

Transazione ben formata

Una transazione ben formata (a tempo di esecuzione) prevede

- Un inizio
 - `start/begin transaction`
- Una fine
 - `end transaction`
- Nel cui corso viene eseguito solo uno dei due comandi
 - `Commit` O `Rollback`
- E in cui non avvengono operazioni sulla BD successive a questi comandi

Siccome la transazione è un'unità atomica dopo che ho fatto `commit` si prevede che non ci siano altre azioni: la transazione termina lì.

Proprietà ACIDE

- Atomicità
- Consistenza
- Isolamento - non viene influenzata da transazioni concorrenti
- Durata (persistenza)

Il nostro DBMS deve garantire le proprietà acide delle transazioni (tutte quante)

Atomicità



Indivisibile

Data una transazione che ha un inizio e una fine o sono visibili (sulla base di dati) l'inizio e la fine della transazione, oppure nulla.

L'approccio relativo alle transazioni è "o faccio tutto o non faccio niente". Non posso lasciare la transazione a metà in nessun caso: devo gestire i guasti in maniera da sapere se sta succedendo qualcosa durante la transazione.

In questo caso devo fare **UNDO** delle operazioni svolte.

Farò un **REDO** delle operazioni, se è necessario ripeterle.

Esito:

- Commit = caso normale in cui ha effetto la transazione
- Abort (o rollback)
 - Può essere gestito dalla transazione stessa, per un qualsiasi motivo
 - Può essere richiesto dal sistema

Consistenza

La consistenza richiede che l'esecuzione della transazione non violi i vincoli di integrità definiti sulla BD

Quando il sistema rileva che una transazione sta violando uno dei vincoli (es. inserimento di tupla con chiave già presente) il sistema interviene per annullare la transazione o correggere la violazione del vincolo.

- La verifica dei vincoli di integrità può essere
 - Immediata (nel corso della transazione)
 - Differita (alla conclusione della transazione)
- La consistenza garantisce che:
 - se lo stato iniziale è corretto
 - anche lo stato finale è corretto
 - lo stato finale è corretto se va a buon fine e applica in maniera corretta e consistente quello che ho effettuato con la transazione.
 - lo stato finale è corretto se non va a buon fine se mi troverò alla fine in uno stato uguale a quello iniziale.

Isolamento

L'isolamento richiede che l'esecuzione di una transazione sia indipendente dall'esecuzione di altre transazioni.

Il sistema deve fare in modo che non ci sia dipendenza tra l'esecuzioni concorrenti.

L'esecuzione in parallelo deve darmi lo stesso risultato di quelle sequenziali.

Il sistema quindi esegue le transazioni in maniera isolata.

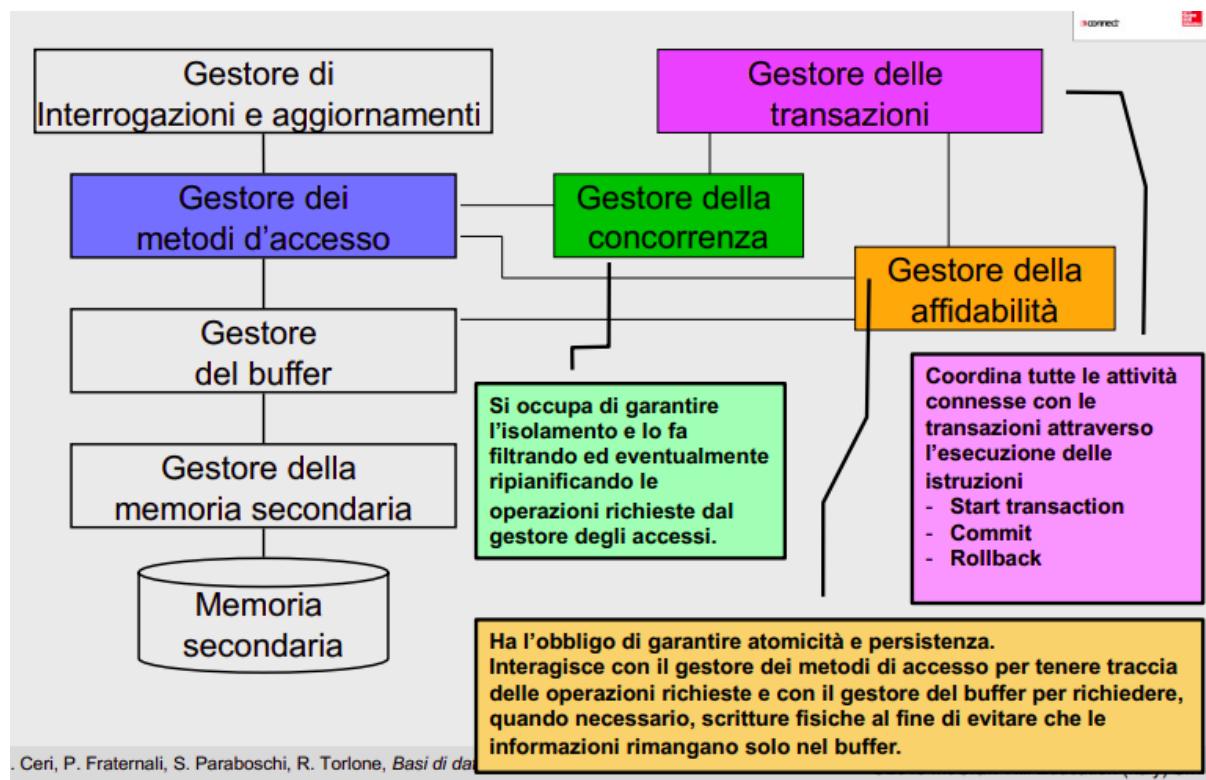
Persistenza

Un DBMS deve garantire che nessun dato, nessuna operazione, che viene eseguita sulla base di dati, venga persa per nessun motivo.

- Questo è garantito sia in caso di qualsiasi tipo di guasto che in condizioni normali
- Un Commit significa impegno

Moduli e proprietà

Il DBMS ha dei moduli per garantire tutte le proprietà appena viste:



Gestore dell'affidabilità

È quello che assicura atomicità e persistenza, che l'esecuzione delle transazioni avvenga in maniera corretta (si parte da uno stato iniziale corretto e finisce in uno stato finale corretto)

- Devono essere garantite qualunque cosa succeda.
- Si può verificare un guasto, qualche errore, ecc ma è garantita correttezza e consistenza
 - A fronte del verificarsi di un guasto utilizza delle operazioni di **ripristino** (**recovery**) abbiamo due meccanismi:

- Ripresa a caldo
- Ripresa a freddo
- Gestisce l'esecuzione dei comandi transazionali.

Usa il **LOG**:

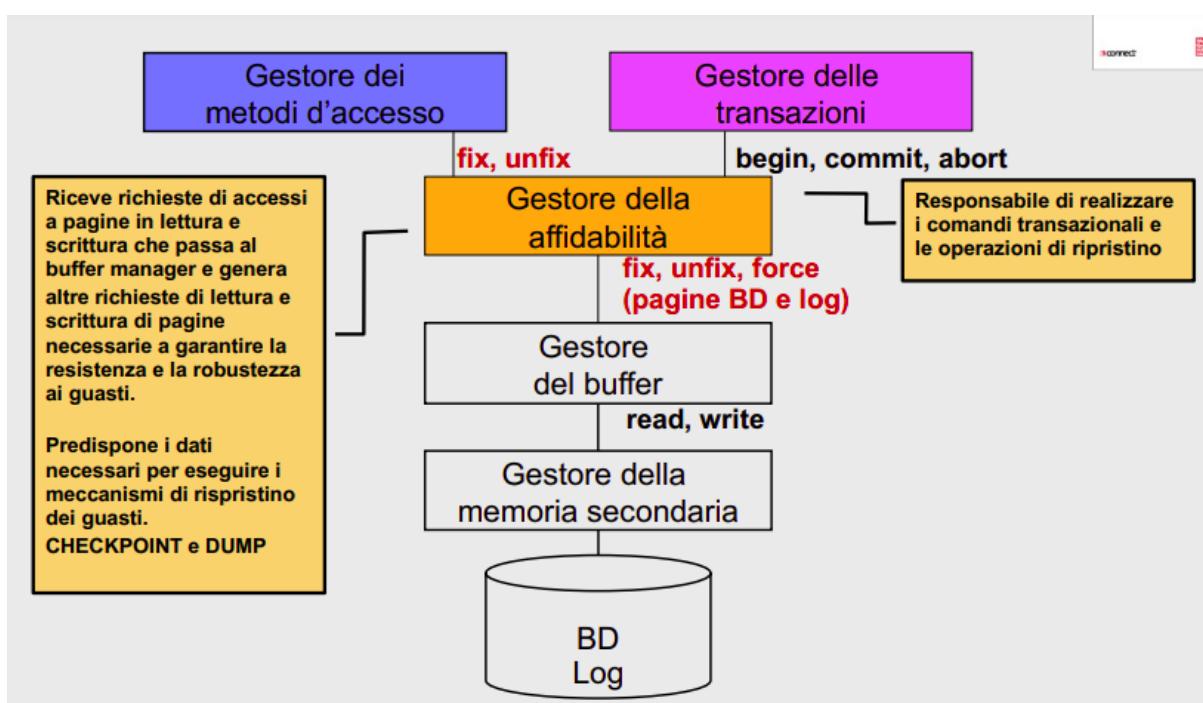
Il log viene anche chiamato giornale / diario di bordo - è un archivio permanente che registra quello che succede nel DBMS. Perdere il LOG è una **catastrofe**: perdo tutto ciò che ho fatto.

Ogni azione di scrittura sulla base di dati viene protetta tramite un'azione sul log.

Questo mi permette, se devo tornare indietro cosa disfare, nel caso di dover rifare, cosa rifare esattamente.

Non viene cancellato nulla dal Log.

Architettura del gestore dell'affidabilità:



Persistenza delle memorie

Devo avere una memoria che risulti persistente ai guasti: il file di Log dovrebbe essere memorizzato in una memoria che non può danneggiarsi.

- Memoria centrale: non è persistente (ci serve per altro)
- Memoria di massa: è persistente ma può danneggiarsi (es. del disco)
- Memoria stabile: memoria che non può danneggiarsi
 - L'idea che non possa danneggiarsi è ovviamente un'astrazione, perché nessuno può garantire che la probabilità di danneggiarsi sia nulla, può sempre succedere qualcosa per cui la memoria viene persa
 - Come riduco al minimo questa cosa? Attraverso la ridondanza:
ho diverse copie su diverse memorie stabili, che mi garantiscono una maggiore probabilità di mantenere al sicuro l'informazione.

Un guasto alla memoria stabile è considerato **catastrofico**.

Log



È un file sequenziale gestito dal gestore dell'affidabilità e in memoria stabile

- In sequenza raccoglie quello che succede nella mia Base di Dati, tenendo traccia di tutto
- È scritto in memoria stabile, quindi sicura il più possibile
- Riporta tutte le operazioni in ordine
- I record che vanno nel file di log sono di due tipi
 - Descrizione delle operazioni delle transazioni
Registro in ordine ogni attività svolta all'interno delle transazioni.
 - Record di sistema
Sono record che indicano l'effettuazione di operazioni specifiche (`DUMP` - backup della base di dati)

Operazioni delle transazioni

- begin, `B(T)` - T è identificativo della transazione
- insert, `I(T, O, AS)` - O è l'oggetto coinvolto, AS è il valore dell'oggetto dopo la modifica AFTER STATE
- delete, `D(T, O, BS)` - BS è BEFORE STATE, quindi il valore prima della modifica
- update, `U(T, O, BS, AS)`
- commit, `C(T)`
- abort, `A(T)`

Record di sistema

- `dump`
Indica che ho fatto un backup, una copia, fatto di solito quando la base di dati non è operativa
- `checkpoint`
È un'operazione svolta periodicamente che ha l'obiettivo di registrare le operazioni attive in quel momento

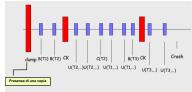
Struttura del log

Il log serve a ripristinare le informazioni, i checkpoint e i dump ci aiutano a ricostruire la storia della mia base di dati, per poter ripartire da determinati punti. I record di sistema agevolano le operazioni di ripristino.

La linea è **temporale**:

In rosso abbiamo
record di sistema e in
blu le transazioni.

Ipotizzando che si verifichi un *crash* so che posso ripartire dalla copia ed eseguire le successive operazioni.



A seconda del tipo di guasto che si verifica il DBMS dovrà cercare di risistemare la situazione, i meccanismi da mettere in atto cambieranno a seconda del tipo di problema.

Abbiamo bisogno di operazioni specifiche che ci permettano di disfare (**UNDO**) e rifare (**REDO**) le operazioni.

Cosa faccio?

Undo di una azione su un oggetto O:

- *Update, Delete*
Ricopiare il valore del before state (BS) nell'oggetto O.
- *Insert*
Eliminare O

Redo di una azione su un oggetto O:

- *Insert, Update*
Ricopiare il valore dell'after state (AS) nell'oggetto O
- *Delete*
Eliminare O

Checkpoint



È l'operazione utile a fare il punto della situazione: mi serve per le operazioni di ripristino.

So che in quel momento lì ho una serie di transazioni attive su cui devo concentrarmi nella fase di ripristino.

- Registro quali transazioni sono attive in quel momento (è avvenuto il begin ma non si sono concluse)
- Confermo quelle che sono terminate e quelle che non sono iniziate (mi interessa meno)

Modalità:

- Per un momento sospendo l'accettazione di richieste di ogni tipo (congelo la situazione), trasferisco in memoria di massa tutte le pagine sporche che ancora non erano inserite in memoria di massa.
- In questo momento so che quella modifica è permanente e definitiva
- Registro sul log gli identificatori delle transazioni in corso

- Riprendo l'accettazione delle operazioni.

Non può succedere che spegnendo la memoria centrale, perdo le modifiche. Le transazioni a metà strada sono anch'esse elencate nel checkpoint.

Dump

Copia, back up della base di dati. Si memorizza in memoria stabile.

Vi si aggiungono tutti i dettagli pratici: che file, che dispositivo, ecc. ecc.

Regole di base

Regole seguite dal gestore delle transazioni che permettono di ripristinare la correttezza della base di dati a fronte di guasti.

1. Write Ahead Log (WAL)

Scrivere sul log (parte before) prima del database.

Mi accerto in questo modo che nel file di Log, in caso di necessità di disfare l'operazione ho il valore precedente. Questo tipo di regola fa sì che per ogni tipo di aggiornamento eseguito sulla base di dati, ho reso disponibile il valore precedente la scrittura.

2. Commit-Precedenza

Si scriva nel log la parte after prima del commit.

Vuol dire che scrivo nel file di log perché voglio sempre sapere qual è la cosa che dovrebbe essere definitiva. In caso di malfunzionamento quindi posso rifare le operazioni, avendo mantenuto il nuovo valore.

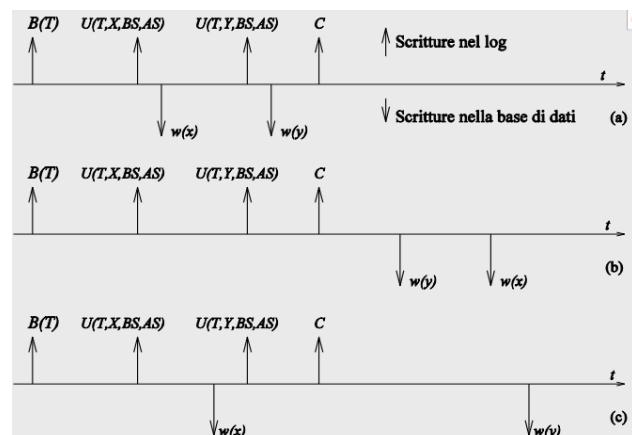
Sto cercando di regolamentare il momento in cui scrivo nella base di dati. In realtà però ci sono varie alternative su quello che faccio:

Tecnica prima il record di Log e poi il record nella BD

Tecnica di aggiornamento immediato

Una cosa più generale è non imporre nessuna delle due alternative: le scritture sulla BD, una volta protette dalle opportune scritture sul Log, possono avvenire in un qualunque momento rispetto alla scrittura del record di COMMIT nel Log

- Consente al Buffer manager di ottimizzare le operazioni

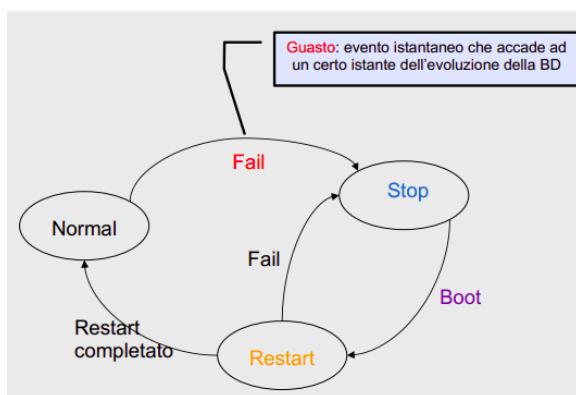


Tutti e tre gli schemi rispettano le regole WAL e CP e scrivono i record di COMMIT in modo sincrono.

Si differenziano per il momento in cui scrivono le pagine.

Classificazione dei Guasti

	Cosa succede?
• Guasti di Sistema (Guasti Soft) Errori del programma, crash, ecc.	Perdo la memoria centrale (le operazioni che stavo facendo in quel momento) ma non quella secondaria Effettuo: ripresa a caldo
• Guasto di Dispositivo (Guasto Hard) Perdo qualcosa sui dispositivi di memoria secondaria	Cosa succede? Potrei perdere la base di dati, prevediamo di perdere la memoria secondaria ma non quella stabile. Effettuo: ripresa a freddo



Modello Fail-stop

Quando il sistema individua un guasto, qualsiasi esso sia, immediatamente forza un arresto delle transazioni. Dopo di che ripristino il corretto funzionamento del sistema operativo (reboot).

- Procedure di **ripresa a caldo**
- Procedura di **ripresa a freddo**

Al termine il sistema è di nuovo utilizzabile

Obiettivi della procedura di ripresa:

- Classifico le transazioni
 - Completate
 - Potenzialmente attive
 - In commit - può servire il **REDO** delle operazioni
 - Senza commit - devono essere annullate con un **UNDO**

Nel file di Log avrò la storia in cui so se una certa transazione andrà in commit oppure no.

Ripresa a caldo

Utilizzo il file di Log e prevede 4 fasi:

1. Ripercorro a ritroso il file di log, fino a che arrivo all'ultimo record di checkpoint, in cui ho fatto una verifica della situazione
2. Costruisce due insiemi di transazioni
 - Insieme di **Undo**
 - Insieme di **Redo**

Per costruire gli insiemi inizializza l'**undo** con le transazioni attive al checkpoint, almeno inizialmente, inizializza l'insieme **Redo** come insieme vuoto.

Dopo di che ripercorre il log in avanti:

- aggiunge in **Undo** tutte le transazioni per cui è presente un record Begin
 - e sposta da **Undo** a **Redo** le transazioni per cui è presente un Commit
3. Ripercorre il log all'indietro, fino alla più vecchia azione delle transazioni in **Undo** e **Redo**, disfaccendo tutte le azioni in **Undo**
4. Ripercorro ora in avanti facendo quelle in **Redo**

// Riga guardare bene la parte di lezione con l'esempio

Ho quindi usato il file di log per capire quali erano le operazioni da fare e disfare per ripristinare la situazione corretta.

Ripresa a freddo

Ho bisogno di sfruttare i backup memorizzati nel file di Log.

1. Ripristino la base di dati a partire dal backup
Devo quindi per prima cosa accedere al più recente record di DUMP
2. Eseguo le operazioni registrate sul Log fino all'istante del guasto
Si eseguono tutte quante, quindi rifaccio tutto quello che ho perso
3. Applico la **ripresa a caldo**.

Questo garantisce persistenza e atomicità relativamente all'istante del guasto: arrivo all'ultimo punto consistente della "vita precedente" della base di dati.

Controllo di concorrenza

Un DBMS spesso deve servire diverse applicazioni e rispondere alle richieste di diversi utenti che a volte accedono alle stesse informazioni. Essendo il numero di transazioni molto elevato, è indispensabile che le operazioni avvengano eseguite Concorrentemente, la loro esecuzione Seriale è impensabile, perché ci impiegheremmo un tempo troppo lungo, creando code infinite.

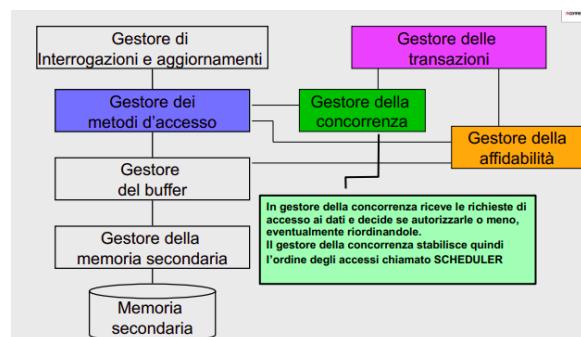
A fronte di richieste concorrente ai dati, il Gestore della concorrenza deve garantire che non si verifichino problemi.

In particolare vedremo che le operazioni vengono schedulate in modo da essere messe in ordine, stabilito dal gestore della concorrenza e chiamato **schedule**.

Unità di misura:

- TPS (Transaction Per Second)
- Unità di misura utilizzata per caratterizzare il carico applicativo di un DBMS

Controllare la concorrenza significa evitare che si verifichino anomalie causate dall'esecuzione concorrente.



Modello di riferimento:

Describe le operazioni che vengono fatte dalle transazioni, come operazioni di input-output su oggetti astratti, x, y e z.

In particolare le vediamo come operazioni di lettura e scrittura, quello che il gestore fa è accettare o rifiutare sequenze di gestioni, per evitare le anomalie.

Anomalie

- Perdita di aggiornamento
- Devo garantire che il risultato debba essere lo stesso di quello che sarebbe stato eseguendo tutto in maniera seriale.
- Lettura sporca
- Lettura inconsistente
- Aggiornamento fantasma
- Inserimento fantasma

Anomalie delle transazioni ricorrenti

Cosa succede se vengono a mancare le proprietà acide?

Anomali = l'esecuzione di due o più transazioni non dà lo stesso risultato dell'esecuzione seriale delle stesse. Se ho due transazioni voglio che la loro esecuzione concorrente mi dia lo stesso risultato di quelle seriali. Se questo non succede c'è un problema.

Perdita di aggiornamento

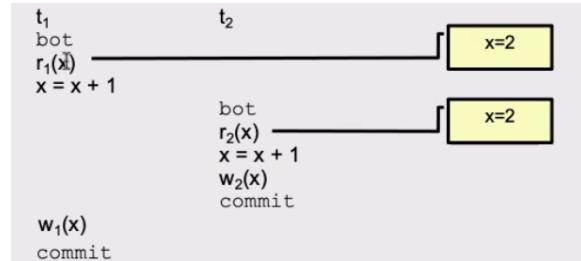
Ipotizziamo di avere due transazioni identiche che operano su x, stesso oggetto della base di dati.

$t_1 : r_1(x), x = x + 1, w_1(x)$

$t_2 : r_2(x), x = x + 1, w_2(x)$

Se ipotizzo che inizialmente avevo $x = 2$, alla fine devo avere $x = 4$.

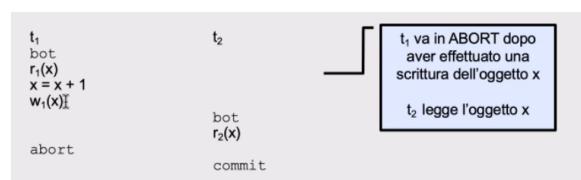
Se le eseguo in maniera seriale questa cosa è garantita: quindi garantisco anche le proprietà acide. Quello che noi pretendiamo dal DBMS è che l'esecuzione concorrente mi dia lo stesso risultato di quella seriale.



Nella versione concorrente quello che succede se ho un errore è che perdo l'effetto della transazione t_2 , perché sovrascrivo il valore calcolato → anomalia: perdita di aggiornamento.

Lettura sporca

La transazione t_1 ha scritto il valore nella base di dati, e questo viene letto dalla transazione t_2 , ma andando la transizione t_1 in abort quello che succede è che avrò comunque 4, mentre invece avendo t_1 abortito dovrebbe essere 3.



Leggo un valore sporco, a cui non dovrei poter accedere.

Ripristinare la correttezza

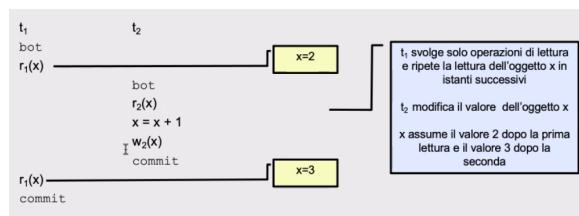
Imporre che a fronte dell'abort di t_1 , potrei imporre l'abort anche a t_2 e anche tutte le transazioni ad esse collegate.

Ovviamente questa cosa potrebbe generare un effetto domino: potrei non avere più controllo su quello che gestisco - questa gestione dell'anomalia è **impraticabile**.

Lettura inconsistente

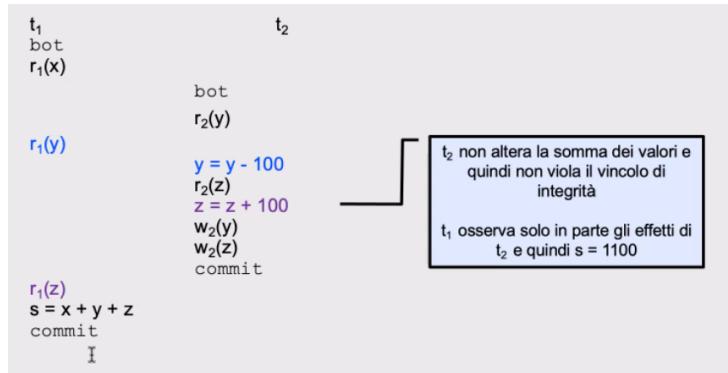
Sarebbe opportuno che una transazione che accede due volte all'attesa BD trovi esattamente lo stesso valore per ciascun dato letto

- la transazione non deve risentire dell'effetto di altre transazioni.



Aggiornamento fantasma

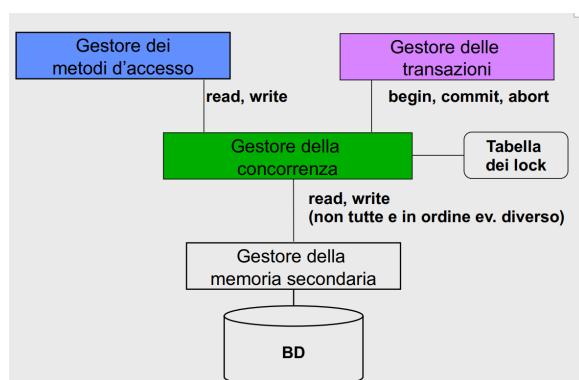
BD con gli oggetti x, y e z che soddisfano un vincolo di integrità $x+y+z=1000$.



Teoria di controllo della concorrenza

Come fa il DBMS a garantire che queste anomalie non si verifichino?

Il DBMS a fronte delle descrizioni delle transazioni stabilirà quando un'esecuzione concorrente è accettabile oppure no. Devo avere un meccanismo che mi permetta di stabilire quale esecuzione concorrente genera anomalia e quando no.



Abbiamo una rappresentazione formale delle transazioni: la descriviamo come una sequenza di azioni di lettura e scrittura

$$t_1 = r_1(x), .r_1(y), w_1(x), w_2(y)$$

Controllo di concorrenza

L'obiettivo del controllo di concorrenza è evitare le anomalie

Cosa facciamo? Rappresentiamo le transazioni tramite schedule, per decidere se possono essere concorrenti o meno.

Scheduler

Lo scheduler analizza gli schedule proposti e decide se accettare, rifiutare o riordinare le operazioni richieste dalle transazioni.

Schedule

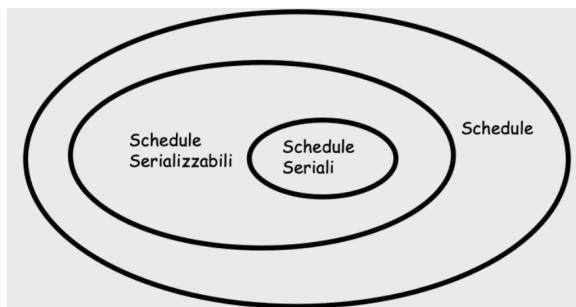
Una sequenza di operazioni input-output di transazioni concorrenti. Queste si incastrano e vengono eseguite nell'ordine temporale che trovo nello schedule.

Quando lo schedule descrive una delle anomalie, quello schedule viene rifiutato: il **controllo di concorrenza** ha la funzione di accettare alcuni schedule e di rifiutarne altri.

Ipotesi semplificativa

Inizialmente assumeremo di sapere l'esito (COMMIT / ABORT), possiamo così ignorare quelle transazioni che vanno in abort → questo ci farebbe concentrare solo sulle operazioni che fanno commit:

- Io schedule chiama una commit-proiezione. Questo ci permette di creare uno schema teorico, ma non posso farlo in pratica perché non conosco l'esito finale delle transazioni.



Schedule seriale

Possiamo dire che lo schedule è seriale se le azioni di ogni transazione compaiono in sequenza

Schedule serializzabile

Produce lo stesso risultato di uno schedule seriale sulle stesse transazioni

Questo cosa vuol dire?

Che lo schedule è serializzabile se produce lo stesso risultato di uno schedule seriale → quindi è anche

Voglio individuare classi di schedule serializzabili che siano sottoclassi degli schedule possibili, siano serializzabili e la cui proprietà di serializzabilità sia provabile a basso costo.

Nozioni di equivalenza

View-equivalenza

Nozione di equivalenza concettualmente interessante, ma inutilizzabile in pratica per ragioni computazionali.



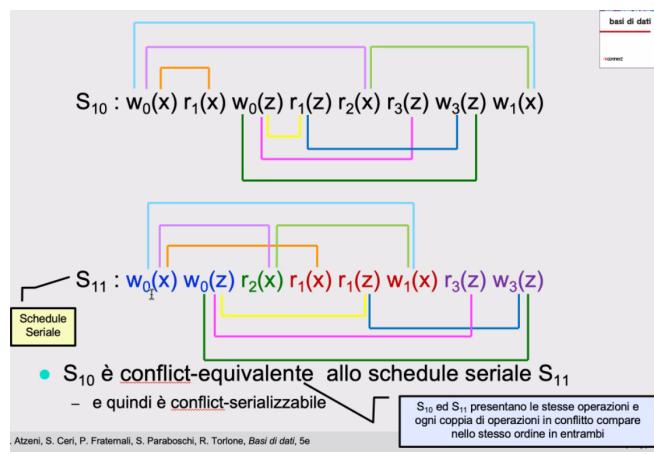
In questo caso S_3 è view-equivalente allo schedule seriale S_4 → è **view-serializzabile**

S_5 invece non è view-equivalente a S_4 , ma è view-equivalente allo schedule seriale S_6 → è **view-serializzabile**

Nell'esame possiamo trovare da due schemi dire se sono **view-equivalenti**

Possiamo dire che due schedule sono **view-equivalenti** quando hanno la stessa relazione legge-da e le stesse scritture finali

Mi interessa definire in particolare la view-equivalenza tra uno serializzabile e uno seriale: posso dire che uno schedule è view-serializzabile se esiste uno schedule seriale view-equivalente ad esso.



Conflict-equivalenza

Si basa sul concetto di conflitto: andiamo a vedere l'equivalenza tra schedule valutandone i conflitti.

Definizioni preliminari

Se due schedule hanno le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi sono conflict-equivalenti

← In questo caso posso dire che sono conflict-equivalenti!

Grafo dei conflitti

Conflict-equivalenza

Si basa sul concetto di conflitto: andiamo a vedere l'equivalenza tra schedule valutandone i conflitti.

Definizioni preliminari

Un'azione a_i è in conflitto con a_j (con $i \neq j$) se queste due azioni operano sullo stesso oggetto e almeno una di queste due operazioni è una scrittura:

- lettura-scrittura
- scrittura-scrittura

Se due schedule hanno le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi sono conflict-equivalenti

Se io ho una schedule che è conflict-equivalente ad una schedule seriale, allora questo è conflict-serializzabile

← In questo caso posso dire che sono conflict-equivalenti!

Anche questo tipo di valutazione di controllo della concorrenza, anche se più rapidamente verificabile, è ancora troppo oneroso: devo calcolare il grafo!

Meccanismo di locking

Il meccanismo che viene utilizzato **nella pratica** si basa sulla protezione per mezzo di un blocco.

Tutte le operazioni di lettura e scrittura sono protette da opportune primitive.

- Tutte le letture sono precedute da **r_lock** (lock condiviso: più transazioni possono leggere contemporaneamente, perché questo non modifica la mia base di dati) e seguite da un **unlock**.
- Tutte le operazioni di scrittura sono precedute da un **w_lock** (esclusivo: non coesiste con altri lock, quando c'è c'è solo lui), seguito anch'esso da un **unlock**

Quando una transazione segue queste regole allora si dice che è ben formata rispetto al lock.

Lock

La transazione può richiedere o un lock esclusivo (anche se è solo in lettura) oppure può chiedere un lock escalation (prima in lettura poi in scrittura)

In base allo stato della risorsa decido come comportarmi e in seguito cambio lo stato della risorsa!

Se la risorsa non è concessa, la transazione richiedente è posta in **attesa** (eventualmente in **coda**), fino a quando la risorsa non diventa disponibile

Richiesta	Esito della Richiesta			Stato assunto dalla risorsa dopo l'esecuzione della primitiva		
	free	r_locked	w_locked	free	r_locked	w_locked
r_lock	OK / r_locked	OK / r_locked	NO / w_locked	free	r_locked	w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked	free	r_locked	w_locked
unlock	error	OK / depends	OK / free	free	r_locked	w_locked

I NO rappresentano i conflitti che si possono presentare
- Richiesta di R o W su un oggetto già bloccato in W
- Richiesta di W su un oggetto già bloccato in R

Dipende dal numero di lettori (contatore)
La risorsa è rilasciata quando il contatore è zero

I tre casi che vediamo sono i casi in cui una scrittura interagisce con un'altra scrittura o una lettura.

Lo stato della risorsa dopo il rilascio dipende da quante transazioni stavano accedendo a quell'oggetto: è come se decrementassi un contatore di lettori - perché è un lock condiviso. Infatti quando libero una risorsa bloccata in scrittura, questa è libera e basta.

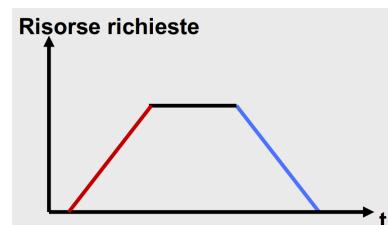
Principio di locking a due fasi

Quasi tutti i sistemi usano i lock in modalità:

- Proteggo tutte le letture e scritture con lock
- Una transazione dopo aver rilasciato un lock non può acquisirne altri

Posso distinguere ora due fasi:

- **Fase Crescente:** acquisisco tutti i lock che mi servono



- **Fase Calante:** Vengono rilasciati i lock acquisiti

Un sistema in cui le transazioni sono ben formati rispetto al lock, in cui il lock manager rispetta la politica della tabella dei conflitti e in cui le transazioni seguono il principio del locking a due fasi il DBMS garantisce la serializzabilità delle operazioni

The diagram illustrates the execution of transaction t1 using the 2PL protocol. The timeline shows the sequence of operations and the state of resources (x, y, z) at each step.

Transazioni		Stato delle Risorse		
t1	t2	x	y	z
		free	free	free
r_lock1(x)			1:read	
r1(x)				
	w_lock2(y)			2:write
	r2(y)			
r_lock1(y)				1:wait
	y=y-100			
	w_lock2(z)			2:write
	r2(z)			
	z=z+100			
	w2(y)			
	w2(z)			
	COMMIT			
	unlock2(y)		1:read	
	r1(y)			
	r_lock1(z)			1:wait
	unlock2(z)		1:read	
	r1(z)			
	s=x+y+z			
	COMMIT			
	unlock1(x)	free		
	unlock1(y)		free	
	unlock1(z)			free

Annotations:

- A callout points to the row where t1 locks y and z: "Le richieste di lock di t1 relative alle risorse y e z vengono messe in attesa e la transazione t1 può procedere solo quando tali risorse vengono sbloccate da t2".
- A callout points to the final row where t1 releases all locks: "Al termine dell'esecuzione la variabile s contiene il valore corretto della somma x+y+z".

018 McGraw-Hill Education (Italy) S.r.l.
87

Versione ristretta del locking a due fasi

è una restrizione del protocollo in cui i lock possono essere rilasciati solo dopo commit o abort

Controllo di concorrenza basato su timestamp

Tecnica alternativa al locking a due fasi che si basa sul timestamp



Un timestamp è un identificatore che definisce un ordinamento totale sugli eventi di un sistema.

Ogni transazione ha un timestamp che rappresenta l'istante di inizio della transazione.

La concorrenza viene gestita facendo sì che vengano serializzate in base al timestamp.

Ad ogni oggetto x associamo due contatori:

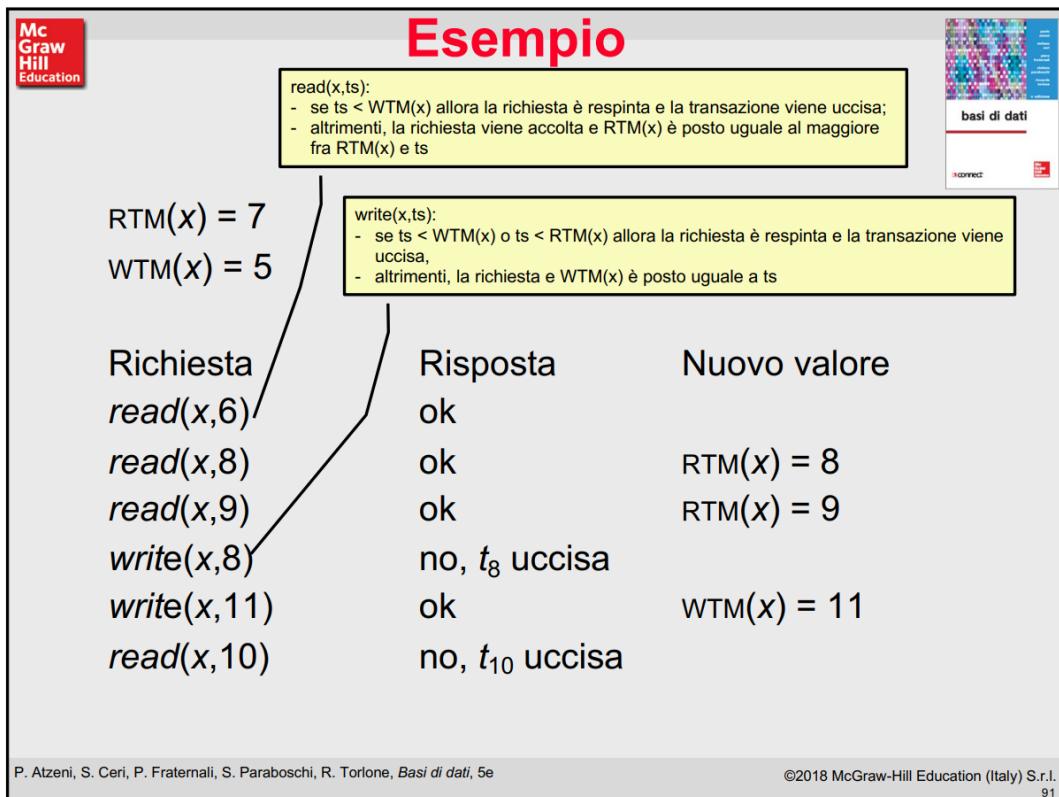
- RTM(x) timestamp della transazione con t più grande che ha letto x
- WTM(x) timestamp della transazione che ha eseguito l'ultima scrittura

Lo scheduler riceve richieste di letture e scritture del tipo:

- `read(x, ts)`
 - se $ts < WTM(x)$ allora la richiesta è respinta e la transazione viene uccisa;

- altrimenti la richiesta viene accolta e viene aggiornato il contatore RTM(x) è posto uguale al maggiore fra RTM(x) e ts
- `write(x, ts)`
 - se ts < WTM(x) o ts < RTM(x) allora la richiesta viene respinta e la transazione viene uccisa
 - altrimenti la richiesta viene accolta e WTM(x) è posto uguale a ts

Quello che succede è che molte transazioni vengono uccise!



Stallo (deadlock) - Blocco critico

Attese incrociate: due transazioni detengono ciascuna una risorsa e aspettano la risorsa detenuta dall'altra.

Nessuna delle transazioni riesce a procedere e il sistema è bloccato!

Per risolvere questo problema abbiamo tre tipi di tecniche:

- Time Out**

Tecnica molto semplice, molto usata nei DBMS commerciali:

Le transazioni rimangono in attesa per un tempo prefissato

- Se passa questo tempo e la risorsa non è ancora stata concessa, allora alla richiesta di lock viene data risposta negativa
- La transazione in deadlock viene tolta dalla condizione di attesa e presumibilmente abortita

- Rilevamento dello stallo**

Controllo il contenuto delle tabelle di lock tutte le volte che lo ritengo necessario.

Il rilevamento di un blocco critico richiede di analizzare le relazioni di attesa tra le varie transazioni e di determinare se esiste un ciclo

- **Prevenzione dello stallo**

Cerco di far sì che non si verifichi, ho più possibilità:

1. Richiedere il lock di tutte le risorse necessarie alla transazione (che però non è detto che io sappia quali sono)
2. Le transazioni acquisiscono un timestamp e si verifica l'attesa solo se si verifica una certa condizione
3. Si basa sull'osservazione che i deadlock si verificano nell'escalation del lock, non si permette l'escalation, ma si introduce l'**update**, un nuovo tipo di lock che è una lettura su cui poi voglio scrivere.

Update

Il lock in scrittura è vincolante, mentre in scrittura altre transazioni possono chiederlo in lettura. Con questo Update Local viene richiesto e acquisito solo da transazioni che vogliono prima leggere e poi scrivere: è unico, quindi blocca su quell'oggetto gli altri update ma non la lettura. Se ci sono blocchi in lettura, la transazione dovrà attendere che abbiano terminato di leggere.

Il linguaggio XML

Un linguaggio di marcatura importante per gestire informazione semistrutturata: sono dati per cui una struttura esiste, ma non necessariamente è stata definita.

Il linguaggio è stato proposto dal w3c e permette di definire un insieme di marcatori che permettano di descrivere il contenuto con tag (marcatori) semplici e leggibili.

Se uso un linguaggio di marcatura, i tag viaggiano insieme al contenuto: il tag mi permette di marcare e dare un significato semantico al contenuto.

Esempi:

- pagine web
- scambio di dati elettronici
- grafica vettoriale
- messaggi vocali



XML consente di descrivere i dati in modo semplice e gestibile

XML è spesso usato come linguaggio di scambio: posso estrarre i dati e caricarli in un contesto applicativo perché permette di dare una **rappresentazione semplice e flessibile dei dati**, è indipendente dal tipo di piattaforma hardware e software utilizzati.

Un esempio:

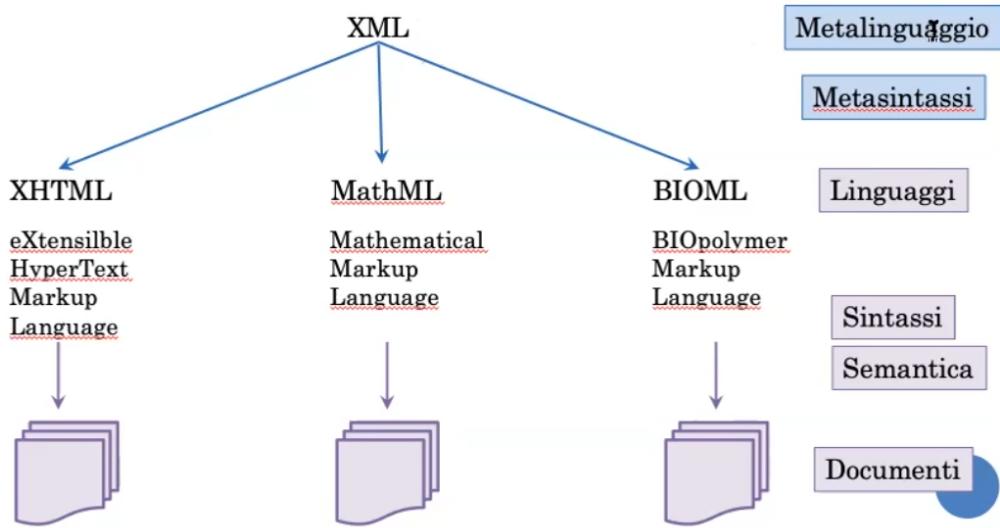
Tutto è all'interno del documento: i marcatori XML sono immersi nel



testo che marchiamo. Inoltre il linguaggio è fatto in modo che sia intuitivo per un essere umano che anche non conosce il linguaggio di markup.

Metalinguaggi

Mi permette di definire un insieme di regole attraverso cui posso definire formalmente altri linguaggi.



Quello che otterrò sono documenti in XML che contengono marcatori che descrivono un determinato tipo di contenuto, specifici per quel tipo.

Per definire un **linguaggio** è necessario un *meccanismo che vincoli l'utilizzo dei marcatori all'interno dei documenti*

- Si deve poter stabilire quali marcatori possono essere utilizzati e come, secondo una precisa struttura logica.
- Bisogna definire una **grammatica**.

Una **grammatica** è un insieme di regole che indica quali vocaboli (elementi) possono essere utilizzati e con che struttura è possibile comporre frasi (documenti). La struttura è vincolante!

XML è un metalinguaggio che mi permette di usare dei marcatori, che possono essere definiti ad hoc per indicare un contenuto preciso.

Quando ho una grammatica posso affermare se un documento XML rispetta o meno una grammatica: è valido o non è valido.

Regole metasintattiche

Posso anche valutare però se un documento XML rispetta le **regole metasintattiche** di XML, si dice che è un documento **ben formato** nel caso.

Un documento è ben formato se:

- ha una sola radice
- ha i marcatori innestati correttamente (il primo che si apre è l'ultimo che si chiude, non ci sono incroci)
- i valori degli attributi sono specificati tra virgolette

In XML posso avere delle **regole semantiche** che dicono cosa posso scrivere in un documento:

un documento che rispetta sia le regole sintattiche che quelle semantiche è **valido**

Regole:

Un documento può essere:

- Ben formato ma non valido

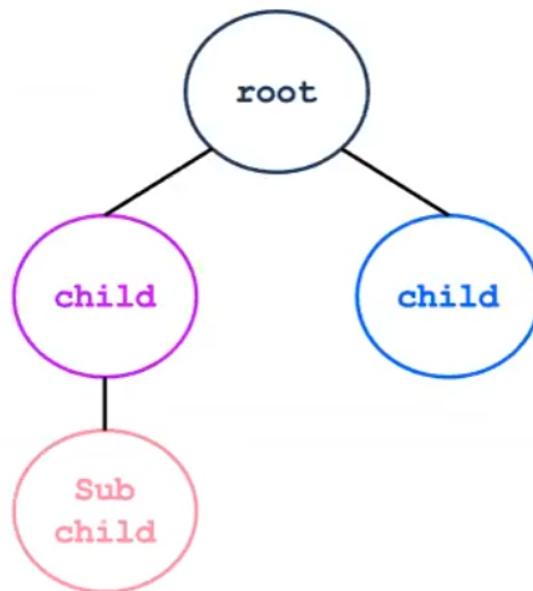
Un documento **NON** può essere:

- Valido ma non ben formato (ovviamente rispetto a una certa grammatica)

Esempio:

```
<root>
  <child>
    <subchild>
    ...
    </subchild>
  </child>
  ...
  </child>
</root>
```

Rispetto a questa struttura ha una corrispondenza in una rappresentazione ad albero: document-tree →



Struttura

Un **documento XML** è:

- strutturato in modo gerarchico
- composto da una serie di elementi

Ogni **elemento** può contenere:

- un frammento di testo
- altri elementi

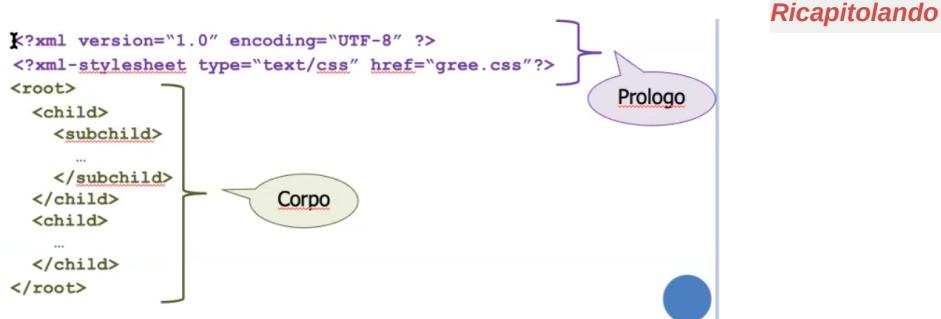
- deve contenere un tag di apertura e un tag di chiusura

Gli elementi di un documento XML sono rappresentabili come un **albero**.

Attributi

Ad un elemento possono essere associate informazioni descrittive chiamate **attributi**. Di solito rappresentano una caratteristica identificativa.

```
<persona cod_fiscale="RSSMRA65E25L781T">
<libro ISBN="A123" titolo="La storia infinita">
```



Un documento XML è costituito da due parti:

- il **Prologo** che contiene una dichiarazione XML e il riferimento opzionale ad eventuali altri documenti o direttive di elaborazione
- il **Corpo** è quello strutturato come già visto

Nel prologo se voglio che un documento sia valido rispetto a una certa grammatica devo specificare la Doctype Declaration (il documento che contiene la grammatica che voglio rispettare)

La grammatica può essere contenuta in un file locale:

```
<!DOCTYPE book SYSTEM "book.dtd">
```

Oppure può essere accessibile tramite un URL pubblico:

```
<!DOCTYPE book PUBLIC "http://www.books.org/book.dtd">
```

Commenti:

posso avere dei commenti nel formato `<!-- Questo è un commento -->`

Elementi e Tag

Un elemento è un frammento di testo racchiuso tra un marcitore iniziale e un marcitore finale. Posso avere testo o anche sotto elementi.

Il primo marcitore è racchiuso tra `<>`, con il nome e la lista di attributi racchiusi.

Il marcitore di chiusura ha lo stesso nome del tag di chiusura ma con davanti lo `<inizio> </inizio>`

Posso avere anche un elemento vuoto! Short end: `<TagName attribute-list />`

Quali nomi posso utilizzare?

Il nome del tag può includere qualsiasi carattere alfanumerico, _, -, . ma può iniziare solo con lettera o

-

Non posso includere altri caratteri di punteggiatura, virgolette, ecc.

XML è casesensitive

Se i nomi non rispettano questa sintassi, non sono **ben formati**



Tipi di contenuto

Di tipo carattere	<code><nome> Mario Rossi </nome></code>
Costituito da altri elementi (figli)	<code><indirizzo> <via> Via Mazzini </via> <civico> 10 </civico> <città> Verona </città> </indirizzo></code>
Misto	<code><dati_anagrafici> Il signor <persona> <nome> Mario Rossi </nome> vive in <indirizzo> <via> Via Mazzini </via> <civico> 10 </civico> <città> Verona </città> </indirizzo> </persona> </dati_anagrafici></code>

Ognuno dei sotto elementi ha come tipo di contenuti: caratteri

Il contenuto di tipo misto è utile se voglio marcare parte del testo con tag XML

Conflitti sui nomi

In documenti complessi bisogna prestare attenzione al dare lo stesso nome ad elementi e attributi diversi:
sarebbe bene evitarlo!

```
<libro>
  <autore>
    <titolo> Sir </titolo>
    <nome> William Shakespeare </nome>
  </autore>
  <titolo> Romeo and Juliet </titolo>
</libro>
```

Possiamo utilizzare quelli che si chiamano Namespace - sono come devi vocabolari di riferimento.

- Si usano prefissi che identificano il vocabolario di appartenenza di elementi ed attributi.
- Ogni prefisso è associato ad un URI (Uniform Resource Identifier) ed è un alias per l'URI stesso.
- L'URI in questione è normalmente un URL: si ha quindi la certezza di univocità.
- L'URI è solo un nome.



Doc ben formati e Doc validi

In XML ci sono:

- *Regole sintattiche*: come dobbiamo scrivere le informazioni all'interno di un documento XML.
- *Regole semantiche* (definite da *grammatiche*): cosa possiamo scrivere in un documento XML.

Un documento XML che rispetta le regole sintattiche si dice **ben formato**.

Un documento XML che rispetta le regole sintattiche e le regole semantiche si dice **valido** (rispetto ad una grammatica).

Come validiamo un documento XML?

Abbiamo due modi:

- Document Type Definition (DTD)
- XML Schema (XSD)

Document Type Definition (DTD)

Voglio specificare un determinato schema o una determinata grammatica: voglio definire l'insieme di elementi e la loro struttura per un determinato documento applicativo.

Un DTD permette di definire una grammatica (definizione degli elementi XML in un determinato contesto e la loro struttura):

- **Il DTD definisce quindi la struttura di un documento valido rispetto a una certa grammatica**
- Descrive quindi i tag ammessi e le regole di annidamento degli stessi.

I DTD vengono utilizzati per la *validazione* dei documenti XML.



Una volta definito il DTD avrò definito le regole semantiche necessarie per validare il mio documento XML: se il documento è **conforme** allora è **valido** rispetto a una certa grammatica.

Dichiarazione del DTD

Per applicare un DTD ad un documento, nel prologo del documento devo inserire una dichiarazione del DTD, seguendo la sintassi:

```
<!DOCTYPE root-element SYSTEM "filename">
```

Impareremo a scrivere il documento DTD.

Esempio:

```
<?xml version="1.0"?>
<!DOCTYPE persona SYSTEM "persona.dtd">
<persona>
  <nome> Barbara </nome>
  <cognome> Oliboni </cognome>
  <email> barbara.oliboni@univr.it </email>
  <telefono> 045 802 7077 </telefono>
</persona>
```

- Questo è un documento ben formato

ESEMPIO: DTD

Gli elementi devono apparire all'interno dell'elemento **persona** nell'ordine specificato

- Cosa deve specificare il DTD

persona.dtd ?

```
<!ELEMENT persona (nome,cognome,email,telefono)>
<!ELEMENT nome          (#PCDATA)>
<!ELEMENT cognome       (#PCDATA)>
<!ELEMENT email         (#PCDATA)>
<!ELEMENT telefono      (#PCDATA)>
```

PCDATA (Parsed Character Data)
è l'unico tipo di dato possibile

Struttura di un DTD



Un DTD è costituito da un elenco di dichiarazioni che descrivono la struttura di ogni elemento.

Le dichiarazioni di un DTD definiscono:

- elementi di un documento
- il modello di contenuto di ogni elemento (specifico se un elemento contiene sottoelementi o stringhe)
- la lista degli attributi

Dichiarazione di elementi

```
<!ELEMENT element_name (content_model)>
<---- oppure ---->
<!ELEMENT element_name category>
```

Il modello di contenuto `content_model` può essere:

- **Children**: elenco dei figli
- **#PCDATA**: contenuto solo testo
- **Misto**: testo e figli, in cui salta l'ordine dei figli

La parola chiave `category` invece può essere:

- **EMPTY**: se l'elemento è di tipo vuoto
- **ANY**: indica che si può inserire testo o elementi qualsiasi purchè dichiarati nel DTD

Modello di contenuto children:

Modello base:



Sintassi: `<!ELEMENT Elemento (E1 , E2 , E3 ,..., En)>`

```
<!ELEMENT persona (nome,cognome,email,tel)>
```

```
<persona>
  <nome> ... </nome>
  <cognome> ... </cognome>
  <email> ... </email>
  <tel> ... </tel>
</persona>
```

Potrei non voler specificare che i figli siano fissi: per questo avere dei figli "alternativi":



Sintassi: `<!ELEMENT Elemento (E1 | E2 | E3 |...| En)>`

```
<!ELEMENT recapito_tel (tel_fisso|tel_mobile)>
```

```
<recapito_tel>
  <tel_fisso> ... </tel_fisso>
```

```

</recapito_tel>
<----- oppure ----->

<recapito_tel>
  <tel_mobile> ... </tel_mobile>
</recapito_tel>

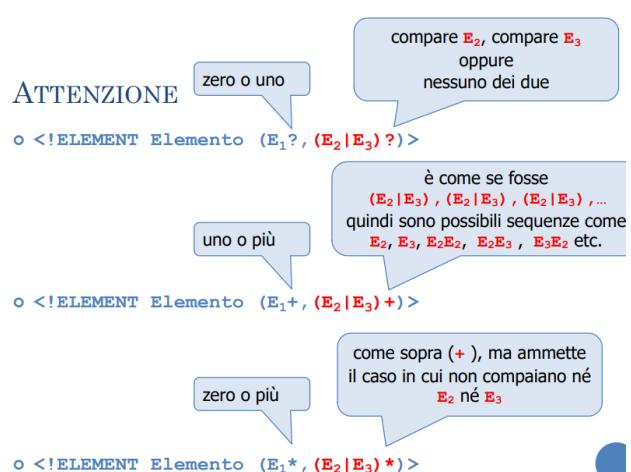
```

Posso anche specificare quante sono le occorrenze di ogni figlio:

- $? = 0 \text{ o } 1$
- $* = 0 \text{ o } +$
- $+ = 1 \text{ o } +$

Esempio:

Si possono innestare sequenze, scelte e operatori usando le parentesi:



Modello di contenuto Mixed:

Specifica che un elemento contiene sia testo che un elenco elementi figli senza ordine specifico

Non posso specificare né l'ordine né la cardinalità.
Importante è che l'elemento #PCDATA deve essere il primo della lista.

```
<!ELEMENT lavoro
  (#PCDATA|incarico|data|compenso)*>
```



Quando si usa XML per dare una struttura difficilmente si usa questo tipo perché praticamente non dà struttura.

Modello di contenuto #PCDATA:

Ci permette di specificare che il contenuto è un insieme di caratteri.



Sintassi `<!ELEMENT Elemento (#PCDATA)>`

```
<!ELEMENT cognome (#PCDATA)>
<cognome>
  Oliboni
</cognome>
```

Modello di contenuto Empty:

Specifica che l'elemento è vuoto. Può avere degli attributi.



Sintassi: `<!ELEMENT Elemento EMPTY>`
 Qual è l'utilità di un contenuto empty?
 Potrebbe contenere ad esempio un
 immagine (data come attributo)

```
<!ELEMENT immagine EMPTY>
<immagine />
```

Modello di contenuto Any:

Specifica che un elemento può contenere:

- Testo.
- Elementi figli (quelli dichiarati nel DTD).
- Contenuto misto.

Equivale a dire che l'elemento ha modello di contenuto misto considerando tutti gli elementi dichiarati nel DTD (è uno shorthand).



Sintassi: `<!ELEMENT Elemento ANY>/`

```
<!ELEMENT pagina ANY>
```

Definire gli attributi

Finora abbiamo visto come dettagliare i modelli di contenuto, ma se io voglio specificare gli attributi per i miei elementi, devo usare un'altra sintassi:

```
<!ATTLIST ElementName
  AttrName1 AttrType1 Value1
  AttrName2 AttrType2 Value2
  ...
  AttrNamen AttrTypen Value_n >
```

Dove:

- `ElementName`: Nome dell'elemento.
- `AttrNamei`: Attributo *i*-esimo.
- `AttrTypei`: Tipo dell'attributo *i*-esimo.
- `Valuei`: Valore di default dell'attributo *i*-esimo o modificatore di presenza.

Tipi di attributo:

Tipo	Significato
<code>CDATA</code>	(Character Data) - Testo
<code>(s₁ s₂ ... s_n)</code>	Enumerazione: valore scelto da una lista enumerativa
<code>ID</code>	Identificatore univoco a livello di documento
<code>IDREF</code>	Riferimento all'attributo di tipo <code>ID</code> di un elemento del documento
<code>IDREFS</code>	Come <code>IDREF</code> , ma ammette riferimenti multipli
...	Altre possibilità (si vedano le specifiche)

Valori di attributo:

Valore	Significato
<code>"VALUE"</code>	Attributo con valore di default pari a <code>VALUE</code>
<code>#REQUIRED</code>	Attributo obbligatorio
<code>#IMPLIED</code>	Attributo opzionale
<code>#FIXED "VALUE"</code>	Attributo con valore fisso pari a <code>VALUE</code>

Tipo CDATA:

Valore di tipo testo

```
<!ELEMENT immagine EMPTY>
<!ATTLIST immagine titolo CDATA #IMPLIED>

<immagine titolo="tramonto"/>
```

Tipo Enumerativo:

Enumerazione: lista di tutti i possibili valori assegnabili all'attributo.

```
<!ELEMENT data EMPTY>
<!ATTLIST data giorno (1|2|3|4|5|6|7|8|9|10|
                      11|12|13|14|15|16|17|18|19|20|
                      21|22|23|24|25|26|27|28|29|30|
                      31) #REQUIRED
      mese (gen|feb|mar|apr|mag|giu|lug|ago|
             set|ott|nov|dic) #REQUIRED
      anno (2005|2006|2007|2008|2009|2010|
            2011) #REQUIRED >

<data giorno="1" mese="ott" anno="2005"/>
```

Tipo ID:

```
<!ELEMENT persona (nome,cognome)>
<!ATTLIST persona cod_fisc ID #REQUIRED>

<persona cod_fisc="RSSMRA65E25L781T">
  <nome> ... </nome>
  <cognome> ... </cognome>
</persona>
```

Tipo IDREF:

Riferimento all'attributo di tipo ID di un elemento del documento.

```
<!ELEMENT persona (nome,cognome)>
<!ELEMENT docente EMPTY>
<!ATTLIST persona cod_fisc ID #REQUIRED>
<!ATTLIST docente ref IDREF #REQUIRED>

<persona cod_fisc="RSSMRA65E25L781T">
  <nome> ... </nome>
  <cognome> ... </cognome>
</persona>
<docente ref="RSSMRA65E25L781T" />
```

TIPO IDREFS:

L'attributo contiene una lista di nomi XML ognuno dei quali deve essere un ID valido di un elemento del documento.

```
<!ELEMENT persona (nome,cognome)>
<!ELEMENT sposi EMPTY>
<!ATTLIST persona cod_fisc ID #REQUIRED>
<!ATTLIST sposi coppia IDREFS #REQUIRED>
```

```
<persona cod_fisc="RSSMRA65E25L781T"> ... </persona>
<persona cod_fisc="BNCFRN63D45L781T"> ... </persona>
<sposi coppia="RSSMRA65E25L781T BNCFRN63D45L781T" />
```

Attributi - DTD

Valori di default

Se abbiamo la necessità di specificare un valore di default, usato nel caso in cui all'attributo non venga assegnato un valore esplicito.



Sintassi:

```
<!ELEMENT elemento >
<!ATTLIST elemento
attributo TIPO
"ValDefault">
```

```
<!ELEMENT immagine EMPTY>
<!ATTLIST immagine larghezza CDATA "0">
<immagine larghezza="100"/>
<immagine /> ----- la larghezza è 0 ----- >
```

Valore #IMPLIED

Si utilizza quando un valore attributo è opzionale e non è possibile stabilire un valore di default.

```
<!ELEMENT contatto EMPTY>
<!ATTLIST contatto fax CDATA #IMPLIED>
<contatto fax="+390458027068"/>
<contatto /> ----- il valore è nullo ----->
```



Sintassi:

```
<!ELEMENT elemento TIPO>
<!ATTLIST elemento fax CDATA
#IMPLIED>
```

Attributo obbligatorio: valore #REQUIRED

Si utilizza quando un attributo è obbligatorio e non è possibile stabilire un valore di default.

```
<!ELEMENT fattura EMPTY>
<!ATTLIST fattura fax CDATA #REQUIRED>
<fattura fax="+390458027068"/>
<fattura /> ----- non si può fare ----->
```

ATTRIBUTI: VALORE #FIXED

Attributi: Valore #Fixed

Si utilizza il valore #FIXED quando un attributo deve avere un valore prefissato. **Non è la stessa cosa di valore di default!**

```
<!ELEMENT mittente EMPTY>
<!ATTLIST mittente dip CDATA #FIXED "Informatica">
<mittente dip="Informatica"/>
<mittente />
<mittente dip="Bioteecnologie"/>----- non si può fare ----->
```

Esercizio

Si modelli un DTD per la descrizione di un linguaggio di marcatura per la gestione di un catalogo. Si rispettino le seguenti specifiche:

- Un catalogo può contenere zero o più libri.
- Un libro è descritto da un titolo, almeno un autore, un editore, un anno ed eventualmente un numero di pagine.
- Un libro è descritto dalle seguenti proprietà: il codice (identificativo univoco), l'eventuale disponibilità del formato ebook (sì/no) per cui si assume di default il valore "no", il tipo di edizione (obbligatoria) che può essere "rilegato" o "brossura", il numero di edizione (opzionale).

```
<!ELEMENT Catalogo (Libro*)>
<!ELEMENT Libro (Titolo, Autore+, Editore, Anno, Numpagine?)>
  <!ELEMENT Titolo (#PCDATA)>
  <!ELEMENT Autore (#PCDATA)>
  <!ELEMENT Editore (#PCDATA)>
  <!ELEMENT Anno (#PCDATA)>
  <!ELEMENT Numpagine(#PCDATA)>
  <!ATTLIST Libro Codice ID #REQUIRED
    Ebook (si|no) "NO"
    Edizione (rilegato|brossura) #REQUIRED
    Numedizione CDATA #IMPLIED>
```

Possibile documento XML:

```
<?xml version="1.0" ?>
<!DOCTYPE Catalogo SYSTEM "Catalogo.dtd">
<Catalogo>
  <Libro Codice='a001' Ebook='si' Edizione='rilegato'>
    <Titolo> NomeLibro </Titolo>
    <Autore> Autore1 </Autore>
    <Autore> Autore2 </Autore>
    <Editore> NomeEditore </Editore>
    <Anno> 2000 </Anno>
  </Libro>
</Catalogo>
```

Soluzione:

```
<!ELEMENT Catalogo (Libro*)>
<!ELEMENT Libro (Titolo,Autore+,Editore,Anno,Npag?)>
  <!ELEMENT Titolo (#PCDATA)>
  <!ELEMENT Autore (#PCDATA)>
  <!ELEMENT Editore (#PCDATA)>
  <!ELEMENT Anno (#PCDATA)>
  <!ELEMENT Npag (#PCDATA)>
  <!ATTLIST Libro
    codice ID #REQUIRED
    ebook (si|no) "no"
    edizione (rilegato|brossura) #REQUIRED
    numero_ed CDATA #IMPLIED >
```

Soluzione di un possibile documento XML **valido**:

```
<?xml version="1.0" ?>
<!DOCTYPE Catalogo SYSTEM "Catalogo.dtd">
<Catalogo>
```

```

<Libro codice="HPPF98" edizione="rilegato">
  <Titolo> Harry Potter e la pietra filosofale </Titolo>
  <Autore> J. K. Rowling </Autore>
  <Editore> Salani </Editore>
  <Anno> 1998 </Anno>
  <Npag> 293 </Npag>
</Libro>
<Libro codice="HPCS99" ebook="si" edizione="rilegato" >
  <Titolo> Harry Potter e la camera dei segreti </Titolo>
  <Autore> J. K. Rowling </Autore>
  <Editore> Salani </Editore>
  <Anno> 1999 </Anno>
</Libro>
</Catalogo>

```

L'elemento libro non ha l'attributo codice obbligatorio!

Conclusioni

Il DTD ci permette di dire qual è la struttura del documento, ma come detto all'inizio è un documento diverso. Non sopporta i namespace, non è possibile vincolare i dati oltre la stringa generica, gli identificatori univoci hanno scope pari rispetto al documento.

XML Schema

Si indica con XSD (XML Schema Definition).



Fa da alternativa al DTD perché è basata su XML, può quindi essere analizzata da un parser XML.

Cosa ci permette di definire?

- Elementi
- Attributi
- Quali elementi sono figli, il loro ordine e numero
- Se un elemento è vuoto oppure contiene testo o altri elementi
- **Tipi di dati sia per gli elementi che per gli attributi**

Gestione dei tipi

XML Schema permette di gestire in modo completo e flessibile i tipi di dati:

- Supporto di tipi di dati primitivi e possibilità di crearne di nuovi.
- Supporto di namespace.
- Supporto di ereditarietà di tipi e di polimorfismo.

È possibile descrivere il contenuto in maniera puntuale. Uso di integer, float, date, string, ...

È possibile lavorare in modo sicuro con dati estratti da database (strong typing).

È semplice la definizione di **restrizioni** sui dati.

- Espressioni regolari, enumerativi, numero caratteri, intervalli numerici, ...

```

<? xml version="1.0" ?>
<messaggio>
  <destinatario> Mario </destinatario>
  <mittente> Ugo </mittente>
  <oggetto> Riunione </oggetto>
  <corpo> Confermata riunione per oggi </corpo>
</messaggio>

```

Come si presenta lo schema XSD di questo XML?

```

<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="messaggio" type="TipoMessaggio"/>
  <xs:complexType name="TipoMessaggio">
    <xs:sequence>
      <xs:element name="destinatario" type="xs:string"/>
      <xs:element name="mittente" type="xs:string"/>
      <xs:element name="oggetto" type="xs:string"/>
      <xs:element name="corpo" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>

```

Il documento conterrà un documento XML che si chiama messaggio e il cui tipo è TipoMessaggio: il tipo è definito sul momento ed è complesso! Sarà formato da una sequenza, formata dai vari elementi che ho inserito successivamente.

A livello di sintassi sembra più complicato rispetto agli altri, a livello di resa però è molto vantaggioso.

Elementi

La radice si chiama **schema**

- Contiene la dichiarazione del namespace

Altre dichiarazioni:

- L'elemento è **element**, di nome **name** e di tipo **type**.
- L'elemento **complexType** mi permette di definire un tipo di nome name.
- L'elemento **sequence** mi permette di identificare la sequenza di elementi figli contenuti in un elemento di tipo messaggio.

XML schema permette di attribuire un tipo sia agli elementi che agli attributi.

- Gli elementi possono avere tipo semplice o complesso
- Gli attributi invece possono avere solo tipo semplice.

Tipi di dati

Tipo semplice (**simpleType**): insieme di stringhe Unicode unito ad una interpretazione semantica di tali stringhe.

- Tipi primitivi: predefiniti nella specifica XML Schema (string, float, integer, date...).
- Tipi derivati: sono definiti in termini di tipi primitivi (derivazione per restrizione).

Tipo complesso (**complexType**): dotato di struttura.

- Definizione di nuovi tipi “da zero”.

- Derivazione per **estensione** o **restrizione**.

Definizione e Dichiarazione

Definizione:

```
<xs:element name= "elementName"
type="elementType"/>
```

Dichiarazione:

```
<xs:attribute name="attributeName" type="attributeType"/>
```

ESEMPIO: DEFINIZIONE INLINE

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="messaggio">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="destinatario" type="xs:string"/>
      <xs:element name="mittente" type="xs:string"/>
      <xs:element name="oggetto" type="xs:string"/>
      <xs:element name="corpo" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Dichiarazione di tipo semplice predefinito

```
<xs:element name="Nome" type="xs:string"/>
<xs:element name="Eta" type="xs:positiveInteger"/>
<xs:element name="DataNascita" type="xs:date"/>
```

Tipi semplici: elementi costitutivi

Un tipo di dato consiste di:

- Spazio dei valori
- Spazio lessicale: rappresentazione dei valori che un certo tipo di dato può assumere
- Insieme di facet: sfaccettature o proprietà che definiscono un tipo di dato

Facet	Vincolo
length	Lunghezza della stringa o numero di elementi della lista
minLength	Lunghezza minima
maxLength	Lunghezza massima
enumeration	Valore enumerativo
maxInclusive	Limite superiore inclusivo (per tipi ordinati)
maxExclusive	Limite superiore esclusivo
minInclusive	Limite inferiore inclusivo
minExclusive	Limite inferiore esclusivo
totalDigits	Numero massimo di cifre (per tipi numerici)
fractionDigits	Numero massimo di cifre frazionarie
pattern	Espressioni regolari
whiteSpace	Controlla la normalizzazione degli spazi bianchi

Tipi semplici derivati

Sono tipi semplici a cui ho modificato lo spazio dei valori per restrizione.

Sintassi:

```
<xs:simpleType name="derivedType">
<xs:restriction base="baseType">
... facets ...
</xs:restriction>
</xs:simpleType>
```

Tipi semplici derivati: intervalli

Usiamo delle facet:

Vanno in **AND** con altre facet sia presenti in una stessa derivazione che presenti in derivazioni successive.

- **maxExclusive**
- **minExclusive**
- **maxInclusive**
- **minInclusive**

Definiscono estremi di intervalli aperti
Definiscono estremi di intervalli chiusi

Definizione di tipo semplice derivato dal tipo predefinito positiveInteger in modo tale che un elemento o attributo dichiarato di questo tipo possa assumere valori compresi tra 18 e 30 (estremi inclusi)

```
<xs:simpleType name="TipoVoto">
<xs:restriction base="xs:positiveInteger">
<xs:minInclusive value="18" />
<xs:maxInclusive value="30" />
</xs:restriction>
</xs:simpleType>

</xs:element name="Voto" type="TipoVoto" />
```

Se ad esempio volessi fare la stessa cosa con le stringhe, definisco una lunghezza fissa con length.

- **length**
- **maxLength**
- **minLength**

Definisce una lunghezza fissa
Definiscono un intervallo di lunghezze

Definizione di tipo semplice derivato "minMaxStringa" in modo tale che un elemento o attributo dichiarato di questo tipo possa contenere stringhe di lunghezza variabile tra 5 e 18

```
<xs:simpleType name="minStringa">
<xs:restriction base="xs:string">
<xs:minLength value="5" />
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="minMaxStringa">
<xs:restriction base="minStringa">
<xs:maxLength value="18" />
</xs:restriction>
</xs:simpleType>
```

Prima definisco il tipo semplice minStringa e definisco quindi la lunghezza minima delle stringhe, in seguito definisco un altro tipo semplice derivato che non va a restringere il tipo di base ma il tipo semplice derivato minStringa.

Tipi numerici

Per definire il numero di cifre complessive e dei decimali nella

```
<xs:simpleType name="maxTre">
<xs:restriction base="xs:decimal">
<xs:totalDigits value="3" />
```

rappresentazione dei tipi numerici si usano le facet:

```
</xs:restriction>
</xs:simpleType>

</xs:element name="Codice" type="maxTre" />
```

Tipi enumerativi

Una enumerazione impone ai valori un insieme finito di possibilità. Si usa la facet:

`enumeration`

- Applicabile a tutti i tipi predefiniti.
- Va in OR con altre enumeration e in AND con altre facet.

Esempio: per il mio simple type le stringhe ammissibili sono ristrette a una certa lista.

```
<xs:simpleType name="lingua">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Italiano" />
    <xs:enumeration value="Inglese" />
    <xs:enumeration value="Giapponese" />
    <xs:enumeration value="Ungherese" />
    <xs:enumeration value="Danese" />
    <xs:enumeration value="Cinese-Mandarino" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="lingua-Indo-Euro">
  <xs:restriction base="lingua">
    <xs:enumeration value="Italiano" />
    <xs:enumeration value="Inglese" />
    <xs:enumeration value="Danese" />
  </xs:restriction>
</xs:simpleType>
```

Tipi pattern

Restringo i valori ammissibili mediante un'espressione regolare:

```
<xs:simpleType name="EuroType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="StrictEuroType">
  <xs:restriction base="EuroType">
    <xs:pattern value="[0-9]*\.[0-9]{2}" />
  </xs:restriction>
</xs:simpleType>
```

- Definizione di tipo semplice derivato che accetta numeri con al più due cifre decimali
- Eventuali simboli speciali nel pattern come `*`, `(`, `)`, `.` possono essere indicati premettendo il `\`
- Il simbolo speciale `*` indica zero o più occorrenze della cifra
- `{2}` indica esattamente due occorrenze

Valori di Default

È possibile specificare un valore di default per un elemento con la sintassi:

```
<xs:element name="nome" type="tipo" default="valore" />
```

Il valore di default viene utilizzato se l'elemento è presente ed è vuoto.

La definizione di vuoto varia in base al tipo di dato nella dichiarazione dell'elemento.

- Il valore deve essere compatibile col tipo di dato.
- In tutti i tipi che ammettono come valore valido la stringa vuota il valore di default non è mai utilizzato.
- I tipi numerici invece non ammettono un valore vuoto e quindi usano il default:

```
<xs:element name="nome" type="xs:integer" default="0" />
```

Valori Fixed

È possibile assegnare ad un elemento un valore fisso con la sintassi:

```
<xs:element name="nome" type="tipo" fixed="valore" />
```

In questo caso:

- Elemento vuoto (come descritto per il default): viene inserito automaticamente il valore fisso.
- Elemento con valore: il valore inserito deve corrispondere al valore fisso.

Attenzione: default e fixed sono mutuamente esclusivi.

Valori nulli

È possibile specificare valori nil con significato identico ai valori NULL nell'ambito dei database.

Nella dichiarazione dell'elemento occorre valorizzare a true il valore dell'attributo nillable .

Nel documento istanza si specifica il valore `nil` valorizzando a `true` il valore dell'attributo `xsi:nil` .

C `xsi` è il prefisso del namespace associato all'URL

<http://www.w3.org/2001/XMLSchema-instance>

```
<xs:element name="taglia" type="xs:integer" nillable="true" />  
  
<taglia xsi:nil="true" /> <---- va bene ---->  
  
<taglia xsi:nil="true">10</taglia> <---- errore ---->
```

ComplexType

Possono avere attributi, elementi figli o contenuto semplice.

Possibilità:

- Contenuto semplice: solo testo e non elementi figli
- Solo elementi figli
- Contenuto misto
- Nessun contenuto

Possiamo definire tipi Anonimi o tipi Con Nome

La definizione inline è di tipo anonimo, se la definizione è separata allora devo dare per forza il nome al tipo.

Definizione con nome

Definizione anonima

```
<xs:complexType name="typeName">  
... tipo di contenuto...  
... attributi...  
</xs:complexType>
```

```
<xs:element name="myElement">  
<xs:complexType>  
... tipo di contenuto...  
... attributi...  
</xs:complexType>  
</xs:element>
```

Solo elementi figli

Posso avere tre sezioni diverse:

- **sequence**

gli elementi dichiarati in questa sezione devono comparire nel documento istanza nell'ordine indicato e con le cardinalità specificate.

- **choice**

nel documento istanza deve comparire uno solo degli elementi dichiarati in questa sezione, con la cardinalità specificata.

- **all**

tutti gli elementi dichiarati nella sezione all possono comparire al più una volta con ordine qualsiasi nel documento istanza.

Oltre alla possibilità di specificare una sequenza in pratica posso anche specificare di scegliere una cosa nella lista di cose proposte, oppure tutti gli elementi dichiarati devono comparire al più una volta, in ordine qualsiasi

Posso specificare anche la
cardinalità! Un certo sotto elemento
ha una cardinalità:

Esempio:

```
<xs:complexType name="TipoSeq">  
<xs:sequence>  
    <xs:element name="e1" type="xs:string"  
        minOccurs="0" maxOccurs="unbounded" />  
    <xs:element name="e2" type="xs:string"  
        maxOccurs="3" />  
</xs:sequence>  
</xs:complexType>  
  
<xs:element name="EsempioSeq" type="TipoSeq" />
```

La cardinalità del primo è 0,n. Quella del secondo è 1,3

```
<EsempioSeq>  
    <e1>aaa</e1>  
    <e1>bbb</e1>  
    <e2>ccc</e2>  
</EsempioSeq>  
  
<EsempioSeq>  
    <e2>ccc</e2>  
</EsempioSeq>
```

Se uso **choice** devo scegliere uno degli elementi dichiarati:

```

<xs:complexType name="TipoScelta">
  <xs:choice>
    <xs:element name="e1" type="xs:string"
      minOccurs="2" maxOccurs="unbounded" /> <--- card 2,n ---->
    <xs:element name="e2" type="xs:string"
      maxOccurs="2" /> <--- card 1,2 ---->
  </xs:choice>
</xs:complexType>

<xs:element name="EsempioScelta" type="TipoScelta" />

```

Cardinalità di gruppo

- I gruppi sequence e choice possono a loro volta avere una cardinalità.
- Si usano sempre gli attributi `minOccurs` e `maxOccurs` inseriti nel tag del gruppo.

Esempio:

La sequenza deve essere ripetuta minimo 2 e massimo 3 volte.

```

<xs:complexType name="TipoSeq">
  <xs:sequence minOccurs="2" maxOccurs="3" >
    <xs:element name="e1" type="xs:string" />
    <xs:element name="e2" type="xs:string" />
  </xs:sequence>
</xs:complexType>

<EsempioSeq>
  <e1>aaa</e1>
  <e2>bbb</e2>
  <e1>ccc</e1>
  <e2>ddd</e2>
</EsempioSeq>

```

Combinazione di sequence e choice

Questo tipo di restrizione non si può fare usando solo le cardinalità, perché non obbligo a fare o uno o l'altro, ma che sono opzionali entrambi.

```

<xs:complexType name="TipoCombinazione">
  <xs:sequence>
    <xs:choice>
      <xs:element name="e1" type="xs:string" />
      <xs:element name="e2" type="xs:string" />
    </xs:choice>
    <xs:choice>
      <xs:element name="e3" type="xs:string" />
      <xs:element name="e4" type="xs:string" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>

<xs:element name="EsempioComb" type="TipoCombinazione" />

```

Esempi di sequenze valide: e1,e4 oppure e2,e3 oppure e1,e3 ecc.

Sezione `all`

Mi permette di dire che quello che sto dichiarando finirà nel documento in ordine qualsiasi, al più una volta.

```

<xs:complexType name="TipoAll">
  <xs:all>
    <xs:element name="e1" type="xs:string" />

```

```

<xs:element name="e2" type="xs:string" />
<xs:element name="e3" type="xs:string"
            minOccurs="0" maxOccurs="1" />
</xs:all>
</xs:complexType>

<xs:element name="EsempioAll" type="TipoAll" />

```

Posso specificare l'opzionalità con MinOccurs e MaxOccurs
(sempre tra 0 e 1)

Tipo mixed

Consente la presenza di caratteri e di elementi.

- Ha senso parlare di contenuto misto (mixed) solo per tipi complessi.
- Per avere un modello mixed è sufficiente indicare nella definizione del tipo complesso l'attributo `mixed` e attribuirgli il valore `"true"`.

```

<xs:complexType name="TipoMisto" mixed="true" >
<xs:sequence>
<xs:element name="Nome" type="xs:string" />
<xs:element name="Indirizzo" type="xs:string" />
</xs:sequence>
</xs:complexType>
<xs:element name="Persona" type="TipoMisto" />

<Persona> Il signor
<Nome> Mario Rossi </Nome> vive in
<Indirizzo>
Via Mazzini 10 Verona
</Indirizzo>
</Persona>

```

Tipo vuoto

Mi basta definire un tipo complesso privo di contenuto:

```

<xs:complexType name="TipoVuoto" >
</xs:complexType>

<xs:element name="immagine" type="TipoVuoto" />

<immagine />

```

Attributi

Gli attributi possono essere contenuti solo da elementi di tipo complexType.

Devono essere dichiarati dopo il modello di contenuto, usando la sintassi:

```
<xs:attribute name="attributeName" type="attributeSimpleType" use="optional|prohibited|required" />
```

Dove:

```

<xs:complexType name="TipoConAtt" >
<xs:sequence>
<xs:element name="e1" type="xs:string" />
<xs:element name="e2" type="xs:string" />
</xs:sequence>
<xs:attribute name="att" type="xs:string" />

```

```

</xs:complexType>

<xs:element name="EsempioAtt" type="TipoConAtt" />

<EsempioAtt att="xyz">
  <e1>aaa</e1>
  <e2>bbb</e2>
</EsempioAtt>

```

Che senso ha definire un attributo proibito?

Immaginiamo di definire un tipo complesso che definisca il tipo di una forma: es. lunghezza, larghezza, altezza, ecc.

Se poi definisco un tipo complesso di tipo quadrato, gli attributi raggio e diametro non devo usarli: dato che box ha tipo di base shape, allora devo definire gli attributi raggio e diametro come proibiti.

Anche per gli attributi possiamo dare la definizione inline.

Per quanto riguarda la possibilità di valori di default e fixed, vale quanto detto prima per gli elementi.

Elementi a contenuto semplice e attributi

Gli attributi possono essere dichiarati solo su elementi complessi.

- È però possibile derivare un tipo complesso da un tipo semplice ed estenderlo aggiungendo attributi.
- Per far ciò si utilizza il modello di contenuto simpleContent con questa sintassi:

```

<xs:complexType name="TypeName"> xs:simpleContent
  <xs:extension base="baseType">
    <xs:attribute name="attName" type="attType" />
  </xs:extension>
</xs:simpleContent>
</xs:complexType>

```

Namespace

Uno schema è un insieme di definizioni e dichiarazioni i cui nomi appartengono ad un particolare namespace detto targetNamespace (namespace obiettivo).

Ogni schema può avere un solo targetNamespace. L'attributo targetNamespace va posto nell'elemento schema.

Posso specificare il namespace che sto scrivendo e che tutte le dichiarazioni nel documento fanno riferimento al target namespace.