

Università degli studi di Torino, Dipartimento di Informatica

Esame: *MAABD (LM) (cap. 8-18 -> teoricamente 8, 9, 10, 11, 13, 16, 17 e 18)*

Docenti: *Prof.ssa Maria Luisa Sapino, Prof.ssa Rosa Meo*

Studente: *Ivan Spada*

Indice

PROF.SSA MARIA LUISA SAPINO (6CFU)	2
NOTE INIZIALI	2
GLOSSARIO	ERRORE. IL SEGNALIBRO NON È DEFINITO.
INTRODUZIONE	3
ARCHITETTURA DEL DBMS	7
STORING DATA: DISKS AND FILES	10
OPERATIONS TO COMPARE	29
BUFFER MANAGEMENT	32
QUERY EVALUATION	44
 PRESENZA: 24/05/21	 68
 PRESENZA: 28/05/2021	 77
 ESAME: POSSIBILI DOMANDE CITATE A LEZIONE	 78
 PROF.SSA ROSA MEO (3CFU)	 80
 INTRODUZIONE AI NoSQL DATABASES	 80
MODELLI DI DISTRIBUZIONE DEI DATI E GESTIONE DELLA CONSISTENZA NEI NoSQL DB	82
INTRODUZIONE AL PARADIGMA DI MAP REDUCE	97
RIAK: UN DATA STORE DI TIPO CHIAVE-VALORE	105
HBASE: DATABASE DI TIPO COLONNARE, OPEN SOURCE DI APACHE FOUNDATION	106
MONGODB	110
NEO4J	112
BEFORE SENTIMENTI ANALYSIS: NPL AND PRE-PROCESSING OF TEXTS	115
ANNOTAZIONI SUL PROGETTO	116

Prof.ssa Maria Luisa Sapino (6cfu)

Note iniziali

Due moduli:

- 6cfu (sapino): aspetti architetturali di sistemi per la gestione di basi di dati con riferimento al modello relazionale
- 3cfu (meo): progetto

Esame:

- Meo: esame orale in cui si discute il progetto sviluppato
- Sapino: scritto (domande aperte) (se fatto online, bisogna fare un colloquio -> regola dip.)
- Voto: media pesata dei due voti

Materiale didattico (modulo 2)

- Lucidi presentati a lezione (che saranno disponibili su Moodle)
- Articoli di approfondimento degli argomenti trattati (che saranno disponibili su Moodle)
- Libro di testo: Ramakrishnan and Gehrke, "Database Management Systems", Third Edition, McGraw Hill (aspetti architetturali).

Libro: è consigliato di comprarlo (o comunque reperirlo), mette a disposizione una serie di lucidi.

Modello relazione (basi di dati): degli anni 70 ma è ancora di grande rilevanza perché hanno tanti vantaggi per specifiche applicazioni.

DB noSQL: quelli che vanno oltre le sintassi SQL (usati molto con i big data)

Introduzione

Database: è una collezione di dati organizzati in modo tale da consentire lo svolgimento di certi specifici tasks come il reperimento di informazioni, l'analisi e la visualizzazione. Il DB deve essere organizzato in modo tale da essere facilmente interrogabile in modo NON ambiguo. diversi DB corrispondono a diversi modelli di dati.

Modello dei dati (Data Model): è un formalismo rigoroso che consente di descrivere la struttura dei dati che vanno a popolare un database (ex modello relazionale, ad oggetti OO, basati su grafi ecc.). I modelli descrivono i vincoli dei dati, ex il modello relazionale vincola il modo in cui i dati vengono organizzati come le relazioni, i tipi delle colonne, i vincoli di consistenza, di chiave ecc.

Schema: è un particolare insieme di vincoli sui dati che sono descritti rispetto al formalismo dello specifico data model che si sta considerando e che sono applicati ad una specifica istanza del DB. Nel modello relazione è un insieme di vincoli che dicono quali sono le tabelle, quali sono le relazioni, come sono fatti i vincoli di chiave e le chiavi stesse, ecc. Il fatto che esistano dei vincoli sull'organizzazione dei dati, semplifica gli aspetti realizzativi (gestione, ottimizzazione -> esistenza di algoritmi esigenti e scalabili), il controllo del rispetto dei vincoli aiuta anche alla consistenza dei dati e quindi alla loro organizzazione e correttezza soprattutto all'aumentare dei dati.

DBMS: sistemi (sw e hw) per la gestione delle basi di dati. Permettono di fare operazioni di memorizzazione, interrogazione e manipolazione. Ogni modello di DB hanno i DBMS che più li caratterizzano.

Tipi di vincoli:

- livello **fisico**: data structures. come si organizzano i dati sul disco che ospiterà la base di dati
- livello **logico**: relazionale, OO, OR. Si avvicina al modello fisico. Esprime vincoli/considerazioni sul livello fisico come: quali strutture per l'organizzazione verranno usati, come verranno fatte le associazioni, le indicizzazioni ecc. Attraverso il modello logico ci si avvicina al modello fisico. Il modello logico colma il gap fra concettuale e fisico. Ci sono livelli logici più vicini al livello concettuale che richiedono maggior sforzo di mapping per avvicinarsi a quello fisico, ce ne sono altri che sono molto più vicino al fisico e non rappresentano tutti i modelli che magari si preferisce avere.
Ex il modello OO è molto vicino al livello concettuale perché ha nativamente dei costrutti vicini al livello relazionale (ad esempio le gerarchie e l'ereditarietà).
Ex il modello relazionale è più vicino al livello fisico: il metodo di organizzazione per record è molto vicino a quello fisico di memorizzazione dei dati su disco (mapping facilitato).
Si pone come mediatore per colmare il gap fra quello di altissimo livello (concettuale) e quello di bassissimo livello (fisico)
- livello **concettuale**: UML, ER, Extendend ER. Formalizzano la specifica dei requisiti a livello di applicazione, si limitano a caratterizzare gli oggetti su cui si dovrà operare senza entrare nelle specifiche. È di altissimo livello e descrivere i concetti e le relazioni fra loro

NOTA: la scelta del livello dipende dal contesto.

Modello gerarchico: inizialmente gestione ad albero dove i discendenti erano collegati in maniera bidirezionale attraverso puntatori. Il reperimento delle info in questi database avveniva navigando attraverso questi puntatori. Navigazione del DB assolutamente procedurale (trovare tutti libri a cui si accede specificando i pattern di accesso, ovvero specificando i puntatori -> .next, .child ex).

Modello reticolare: basato su puntatori e navigazione procedurale da parte dell'utente (come quello gerarchico) però nel modello reticolare non si usano gli alberi ma i grafi quindi si perde il vincolo della struttura ad albero. Come quello gerarchico descriveva gli oggetti in termini di oggetti e relazioni fra loro (che erano stabiliti dai puntatori tra A e B)

NOTA:

- sia il modello gerarchico che il reticolare sono funzionali a specifiche applicazioni ma per realtà diverse (come il sistema bancario) questi tipi di modelli non erano così comodi, quindi inizia lo "scollamento" da livello concettuale
- Domanda esame: sono inoltre troppo vincolati dai puntatori che rendono gli algoritmi poco efficienti. il puntatore è di per sé inefficiente perché rendono gli accessi IMPREVISTI, ovvero si sa che un puntatore punta ma non si sa dove va fintanto che non lo si ha analizzato e si ha raggiunto il valore. Queste imprevedibilità (dal punto di vista sulla gestione del db) hanno effetti pesanti sull'ottimizzatore che riesce ad ottimizzazione che query e l'organizzazione per il fatto che c'è una certa regolarità dei dati (ma così non avviene se ci troviamo in situazioni differenti).
- basati su vincoli molto forti in termini di natura gerarchica che è naturale quando i dati hanno già una natura gerarchica, in caso contrario appare come una forzatura.
- avendo un'iterazione procedurale, l'utente deve essere sufficientemente istruito

Punti negativi:

- vincolo gerarchico: inadatto se i dati non hanno già loro una natura gerarchica
- puntatori che non permettono l'ottimizzazione (ad esempio in natura di stime e ottimizzatore)
- interazione procedurale che vincola l'istruzione degli utilizzatori
- uso pesante del puntatore nell'organizzazione/struttura gerarchica del dato che rendeva imprevedibile la collocazione del dato che si sarebbe reso necessario subito dopo quello a cui si era fatto accesso. Il fatto di poter prevedere permette di fare delle ottimizzazioni. Non potendo fare previsioni, la complessità computazionale (e quindi il numero di accessi) aumenta particolarmente.

Generazioni di DB:

- **Prima generazione:** gerarchico e reticolare
- **Seconda generazione:** relazionale (il modello relazionale è basato su uno schema rigido), ha avuto una rilevanza talmente alta da essere ancora utilizzato (usato ad esempio nei sistemi bancari). Il fatto di avere "schemi stabili", rendono i dati prevedibili e quindi è possibile stimare lo spazio di occupazione, si possono quindi prevedere gli indici dei dati di interesse. La prevedibilità è il principale successo dello schema relazionale perché essere prevedibile significa essere ottimizzabile. Ad esempio, ci sono vantaggi nella logistica ma anche nell'accesso ai dati salvando i dati sul buffer che

si sa già serviranno nell'immediato futuro (ovvero salvando i dati vicini alla posizione della testina). Il poter esprimere in maniera dichiarativa (SQL) le operazioni ai dati che sono dentro il db (piuttosto che conoscere l'accesso al disco) è stato un enorme vantaggio che ha sfruttato l'omogeneità, i vincoli a cui sono soggetti i dati (chiave, tipo, ecc.), con SQL non è il portante la pagina in cui sono salvati i dati ma l'utente si preoccupa solo di **dichiarare** che cosa gli interessa fare.

Grandi proprietà/caratteristiche:

- prevedibilità
- assenza di puntatori
- vicinanza alla visione fisica
- linguaggio dichiarativo
- sono sufficienti pochi operatori dell'algebra relazionale per poter implementare le operazioni. È un punto di forza che siano pochi perché per ottimizzare, ovvero meno solo le alternative e meno ci sono costi nel tentativo di ridurre i costi (con il linguaggio operativo, il sistema di preoccupa dell'ottimizzazione -> DBMS). Nota: il problema dell'ottimizzazione è un qualcosa di più complesso, pur essendo pochi gli operati il problema rimane complesso e sarebbe peggio se fossero di più. Inoltre, ciascuno di questi operatori possono essere implementati fisicamente in maniera diversa (ex diversi modi per implementare la selezione, le join, ecc.). L'ottimizzatore deve ottimizzare l'ordine di esecuzione degli operatori e deve scegliere il tipo di implementazione (ex se in una query ci sono tre proiezioni, non è detto che vengano implementate tutte allo stesso modo. viene influenzato ad esempio dalla presenza di un indice, ecc.). C'è ancora della ricerca sull'ottimizzazione perché risulta essere ancora complesso. Non sempre ottimizzare vuol dire avere la soluzione migliore, il costo dell'ottimizzazione non deve superare il risparmio (**il costo è il tempo, l'unica risorsa che nessuno ci restituirà nella vita**). "fare in un certo modo è mediamente più ottimale che farlo diversamente, ovvero ci si accontenta di scegliere il meno peggio".

Difetti:

- rigidità
- non funziona bene in cluster di macchine
- è meno adatto per dati con già una natura gerarchica

Ex:

- Business App

- **Terza generazione:** modelli ad oggetti, sono i modelli che riflettono due scuole di pensiero (OO vs Object-relational). Sono ancora modelli strutturati ma questa volta lo schema è costruito su oggetti, sistemi sw in cui tutto è gestito mediante classi, oggetti, ereditarietà, ecc. Object-relational afferma invece che si possa avere una base relazionale su cui si possa costruire un mondo ad oggetti. I db ad oggetti hanno un potere computazionale più alto rispetto al modello relazionale, hanno anche il concetto di ereditarietà e si possono prevedere funzioni complesse a piacere.

Nota: OQL (Object Query Language) è un altro linguaggio dichiarativo per accedere ai dati, l'utente non si preoccupa di nuovo dei puntatori ecc. Oracle oggi è un Object-relational, non ha stravolto quello che c'era prima (relazionale) ma da la possibilità anche di effettuare aggiornamenti al mondo ad oggetti senza avere perdite (e senza dover formare di nuovo il personale).

Ex:

- Nuove applicazioni
- OO
- Object-relational

- **Quarta, quinta, sesta, ... generazioni:** chi vuole flessibilità deve essere pronto a **tollerare**, in questo caso si possono usare strutture **semi-strutturate** come l'XML. Naturalmente in questo caso si perde **prevedibilità** perché non si sa con esattezza quanto spazio occupa un dato e quindi dove aspettarsi di trovarlo. In XML, dato che c'è flessibilità di schema, non si sa nemmeno se c'è il dato. Lo schema stesso è oggetto di interrogazione (sono previste anche interrogazioni sulla struttura: "verifica se esiste un percorso per raggiungere un certo tag").

Andando avanti tornano i grafi dove la rappresentazione è a grafo ma il grafo in sé è semanticamente informativo (non è quindi un semplice ritorno al reticolare). Le interrogazioni sono di tipo dichiarativo e non sono una "forzatura".

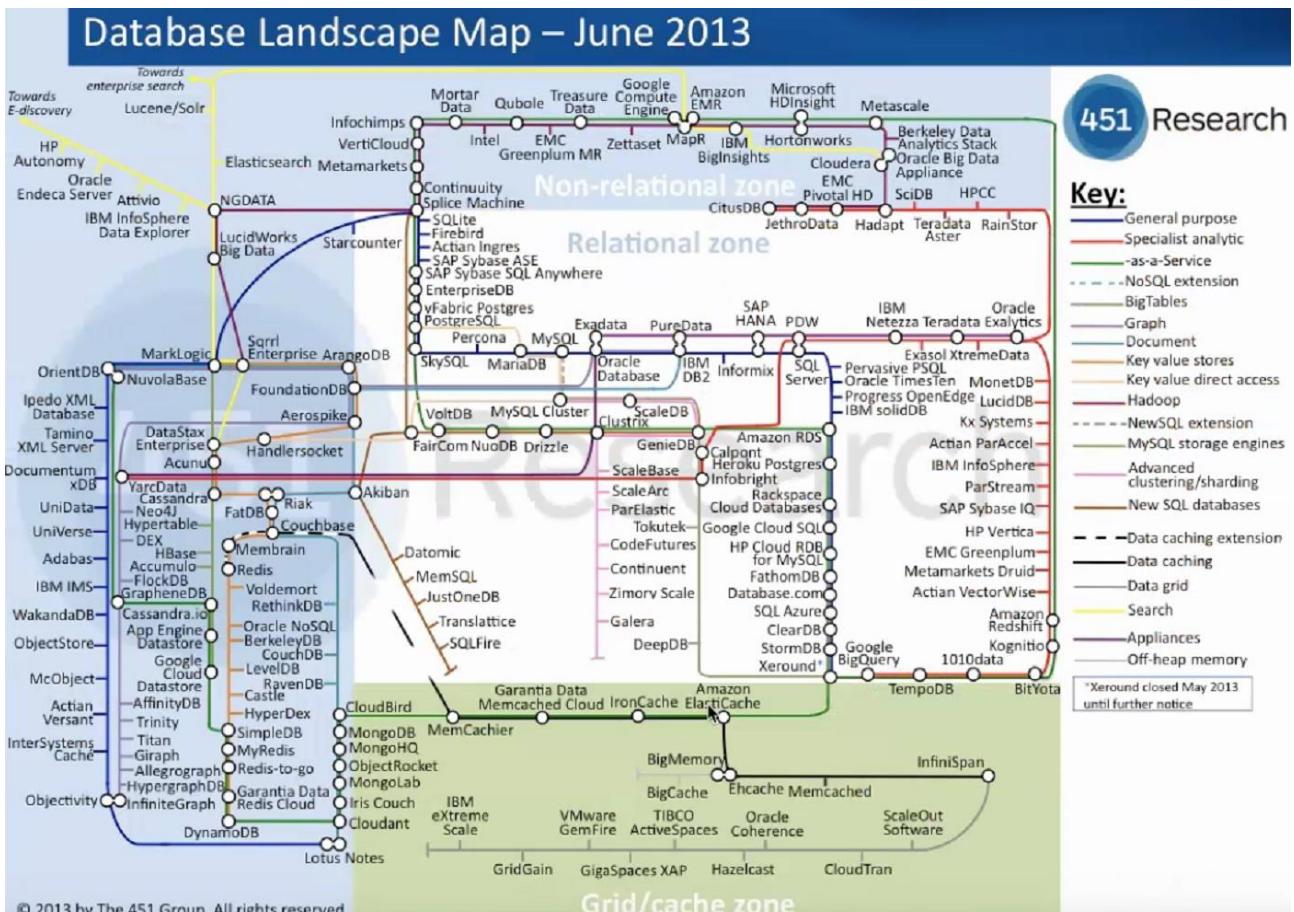
RDF (Resource Description Framework): è il core dei sistemi di rappresentazione della conoscenza per il web semantico, permette di modellare il mondo (complesso a piacere) descrivendo la realtà sotto forma di terze (soggetto, predicato, oggetto). È una rappresentazione a grafo semanticamente informata. Le query prendono il nome di "query di raggiungibilità", ovvero dati due concetti nel grafo deve restituire il percorso fra i due concetti e questa è la relazione tra i due. Cammini diversi corrispondono all'esistenza di relazioni diverse. SPARQL consente di formulare, in modo dichiarativo, le query di raggiungibilità su grafi di conoscenza complessi a piacere.

Tipicamente nei db relazionali vale il concetto di mondo chiuso, ovvero è vero tutto quello che è stato esplicitato nel db (se non è scritto nel db non è vero), nei grafi c'è il principio di mondo aperto ed è possibile rappresentare un certo livello di **incertezza**, ovvero di **confidenza**. In base ai pesi assegnati agli archi è possibile "pesare" il livello di confidenza.

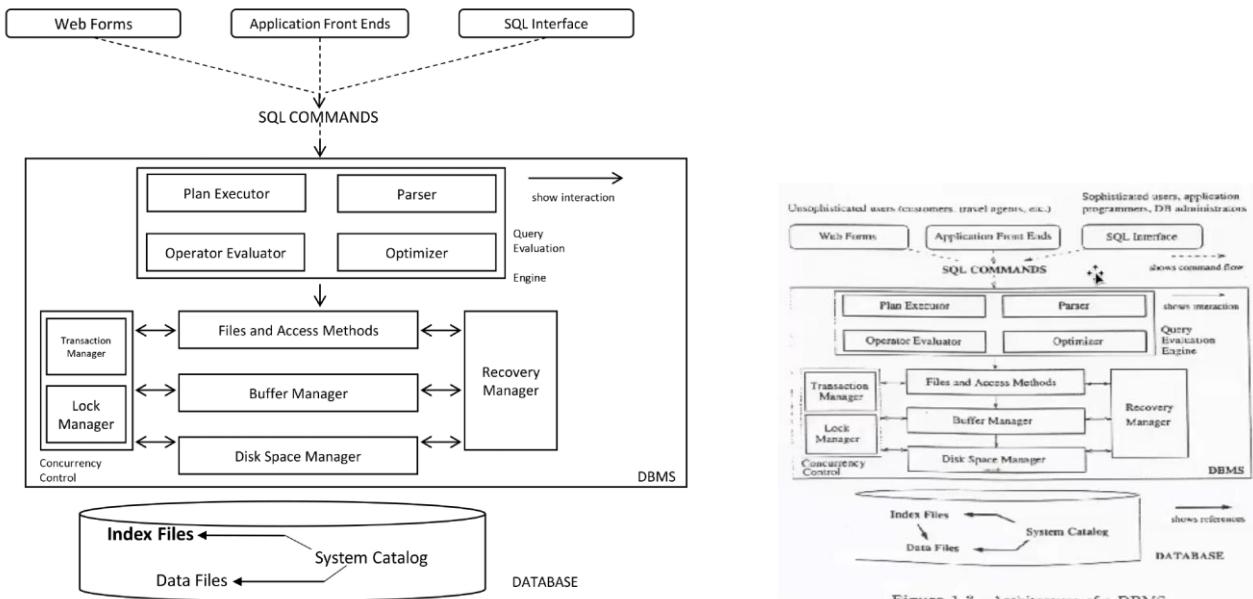
Esempi di app che beneficiano le interrogazioni su grafo: social network (interrogazione sulle relazioni degli influencer), mercato immobiliare (cercare casa in quartieri con determinate caratteristiche), app geografiche. NOTA: le primitive di interrogazione e i dati potrebbero essere diversi da dominio a dominio e quindi ogni applicazione deve avere un'architettura ad hoc. Si usa il modello a grafo quando interessano maggiormente query di raggiungibilità piuttosto che la popolarità di un nodo rispetto ad un altro (quindi lo studio dei cammini e dei nodi che compongono il dato stesso). Nelle banche continua ad andare bene il modello relazionale.

Nota:

- con l'avvento di Internet è cambiato l'accesso e l'utilizzo dei dati



Architettura del DBMS



A livello più basso dell'organizzazione del db, il modulo che agisce direttamente col db è il modulo che gestisce lo spazio disco (**disk space manager**) è il modulo che si occuperà di interagire direttamente con lo spazio disco per gestire la memorizzazione, la lettura e la scrittura dei dati sul disco stesso. Subito sopra c'è ciò che gestisce la memorizzazione e l'accesso ai file, i due interagiscono perché la memorizzazione dei dati avviene alla luce del genere dei file che si stanno memorizzando. In un db relazionale si hanno pochi tipi di file ma memorizzare (i file che rappresentano il disco, i data file(i dati veri e propri inseriti e interrogati, anche chiamati **hip file**), file che è un indice definito su file di dati, catalogo di sistema (che contiene i dati descrittivi relativi a quanto contenuto negli altri file come lo schema del db, l'esistenza di indici, la cardinalità delle relazioni -> contiene ovvero le info utilizzate per le statistiche per valutare le eventuali ottimizzazioni delle queries)).

Access manager: gestisce il controllo dei file e l'accesso ai file stessi.

Buffer manager: l'accesso ai dati contenuti nei file memorizzati su disco, tipicamente non avviene direttamente passando dal file manager al disco ma tipicamente c'è una componente intermedia che è il **buffer manager** (in modo da fare **caching**). Le app non operano quindi direttamente sul disco ma in una pagina che è già in memoria centrale, ovvero che è già nel buffer. Si fa questa scelta perché le operazioni più costose sono quelle di accesso al disco (lettura e scritture su disco, si cerca di limitarle il più possibile). Il **buffer manager** è il modulo che cercherà di sfruttare al meglio l'area limitata di memoria principale riservata al buffer, ovvero dove poter "appoggiare" temporaneamente i dati letti dal disco. Se l'applicazione dell'utente effettua degli aggiornamenti sul record, gli aggiornamenti fatti direttamente sulla pagina del buffer porta la pagina ad essere "potenzialmente" inconsistente rispetto al disco. Se le operazione di modifica vengono effettuate nel buffer, le info nel buffer non saranno più consistenti rispetto a quelle su disco e se si ha un crash di sistema verranno persi i dati (in situazioni di questo tipo subentra il **recovery manager**).

Nota:

- è stato scelto il buffer perché basa sul principio della località temporale quindi tipicamente il dato "serve più volte" e quindi viene "portato in memoria centrale"
- la lettura da disco legge sempre l'intera pagina e riporta il frame nel buffer (non avviene mai la lettura di un solo record).

Recovery manager: si preoccupa di ripristinare una situazione su disco consistente (deve tener traccia quindi di ogni operazione, **file di log**). Utile per ripristinare **file and access manager**, **buffer manager** e **disk space manager** in caso di problemi (come il **crash recovery** e la perdita dei dati in memoria centrale). Per funzionare, il recovery manager deve avere accesso a certe informazioni messe a disposizione dal "**transaction manager**" (chi ha fatto cosa). A differenza del transaction manager, non è sempre attivo ma solo quando serve.

Transaction manager: è sempre attivo (in modo trasparente), costantemente salva ciò che succede nel sistema e gestisce il **file di log** per eventualmente "disfare e rifare" le operazioni che sono state danneggiate dal crash in modo da permettere la fase di **recovery**. Se non fosse efficiente rallenterebbe tutto il sistema quindi le informazioni le salva in memoria principale e non su disco perché in caso contrario rallenterebbe il sistema. La scrittura in memoria centrale viene effettuata al prezzo di rischiare (in caso di crash, qualora manchi corrente) di perdere tutto quello che si era salvato. Verranno quindi utilizzate delle tecniche per

memorizzare opportunamente su disco quanto basta per ripristinare una situazione di consistenza (non necessariamente tutto). Il transaction manager agisce di pari passo insieme al **lock manager** e insieme gestiscono il controllo della concorrenza da parte delle transazioni che accedono eventualmente a dati condivisi.

Lock manager: è un modulo che serve per gestire i casi di accessi concorrenti a uno stesso dato da parte di due o più applicazioni (più processi che vogliono modificare lo stesso dato). Permette gli accessi esclusivi in scrittura ed eventualmente collettivi in lettura. La gestione del **lock** è delicata perché può dar luogo a situazione di **deadlock** (situazione a cui si presenta una attesa circolare) che non sono troppo frequenti ma possono capitare. Un uso eccessivo del lock (soprattutto se è un uso non giustificato e si è “troppo” cauti) rischia di ridurre il potenziale parallelismo e quindi rallenta il throughput di sistema. Limitare drasticamente la parallelizzazione (che in maniera teoria è possibile che generino deadlock ma allo stesso tempo non è nemmeno garantito che lo generino) può avere come effetto collaterale la lentezza del sistema poiché si tende verso la serializzazione del sistema.

Nota: successivamente si vedrà come transaction manager, lock manager e recovery manager lavorano per garantire le (atomicity, consistency, isolation, durability) delle transazioni.

ACID:

- **Atomicity:** più operazioni logicamente distinte operano come fossero un'unica, ovvero vengono eseguite tutte le operazioni all'interno di una transazione oppure nessuna
- **Consistency:** tutte le transazioni che operano su una situazione consistente devono terminare con un'altra consistente
- **Isolation:** le operazioni concorrenti vengono gestite come fossero sequenziali
- **Durability:** deve essere assicurata la persistenza delle operazioni completate

Nota: atomicity e durability sono generalmente affidate al recovery manager (si assicura che tutto venga completato o disfatto e si assicura che tutto quello che viene completato duri nel tempo), consistency e isolation sono affidati al **cconcurrency control manager** (ovvero **transaction manager + lock manager**)

Query evaluation engine: comprende diversi moduli che hanno a che fare con l'interazione dell'utente, ovvero quella che è la valutazione delle query dell'utente. In particolare, si ha un modulo che fa il **parsing** della query (la traduce in equivalenti espressioni dell'algebra relazionale), l'**ottimizzatore** sceglierà fra le diverse espressioni equivalenti (che portano alla soluzione della query) quella più opportuna (ovvero garantisce che la soluzione non sia pessima, ci si limita a espressioni di natura **euristica**). Per ogni operatore si ha algoritmi di ottimizzazione differenti. Il **plan manager** è quello che segue il piano, ovvero la sequenza in cascata degli operatori diversi, ciascun piano verrà valutato dall'**operator evaluator**.

Nota: c'è una forte inter-dipendenza fra moduli del DBMS. Ad esempio, l'ottimizzatore (per poter ottimizzare le queries) deve recuperare le informazioni relative alla memorizzazione dei dati sui dischi e queste info le trova fra i file di catalogo (se appare una domanda di questo genere è importante fare un discorso di progettazione completa). Non si può pensare di frammentare il DBMS e delegare a team diversi lo sviluppo

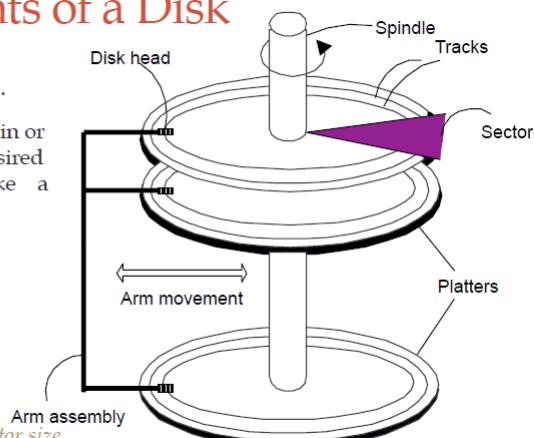
di moduli diversi se non dopo aver ben concordato le specifiche e i comportamenti dei singoli moduli per le parti che sono in interazione.

Storing data: disks and files

Nota: tutti i riferimenti sul libro di testo e i concetti introdotti fanno riferimento alle situazioni in cui c'era il tipico disco rigido che ruota andando ad ignorare i successivi progressi tecnologici. Non ci si preoccuperà di questi punti perché tanto, cambiando i parametri, i concetti di fondo rimangono in genere gli stessi. La componente tecnologica potrebbe cambiare ma quella sw e di progettazione del DBMS rimane pressoché la stessa.

Components of a Disk

- ❖ The platters spin (say, 90rps).
- ❖ The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).
- ❖ Only one head reads/writes at any one time.
- ❖ *Block size* is a multiple of *sector size* (which is fixed).



Disk e files: la memoria secondaria viene rappresentata da un “hard” disk dove vengono salvati in maniera persistente i dati. Le operazioni **read** corrispondono al trasferimento dei dati dal disco alla memoria principale, le operazioni **write** il contrario. Entrambe le operazioni sono particolarmente costose (rispetto a quelle svolte all'interno della memoria centrale) a causa della “lentezza” della memoria secondaria pertanto sono da programmare con cura (studio di risparmio per quando possibile). “Costoso” significa che richiede tempo e quindi rallentano le operazioni di sistema, questo vuol dire che riducono il throughput.

Disks:

- memoria secondaria: struttura fisica in cui vengono salvati i dati in maniera persistente
- vantaggi: random access vs. sequential
- i dati vengono memorizzati e restituiti in unità chiamate “**disk blocks**” o “**pages**”.
- a differenza della RAM, il tempo per restituire una pagina del disco dipende dalla localizzazione sul disco stesso (quindi il tempo di lettura delle pagine può influire sulle prestazioni del sistema a seconda della posizione del dato sul disco)
- Il disco viene visto come una “pila di superfici circolari magnetizzare” che sono in rotazione continua. Essendo le superfici dei dischi magnetizzate, vengono registrate sequenze di bit (0 o 1). In RAM invece la presenza del 0/1 viene registrata come l'assenza o la presenza di un campo elettrico.

- Le testine hanno un unico movimento (interno-esterno, esterno-interno -> come un gira dischi)

Accesso a una pagina del disco: il tempo di accesso (read/write) e un blocco del disco è composto da 3 fasi

- **seek time:** il tempo che comporta lo spostamento della testina per raggiungere la traccia contenente il dato che si vuole andare a leggere (da 1 a 20 msec circa). È la fase più costosa fra le tre ed è quella che si vuole cercare di minimizzare. Dipende da quanto si trova lontana la traccia che contiene il dato rispetto all'ultima che ho letto (quindi dipende dal movimento che il braccio deve fare per raggiungere la traccia). Se il prossimo dato da leggere è sulla stessa traccia, il tempo di seek è zero quindi bisogna cercare di memorizzare opportunamente i dati. **Seek time varia da 1 a 20msec circa.**
- **rotational delay:** il tempo impiegato per attendere che il disco ruoti e quindi la testina raggiunga il blocco (il disco non si ferma se ci fosse uno stop-end-go, l'inerzia fisica per fermarsi e ripartire comporterebbe uno spreco di tempo troppo alto). **Rotational delay varia da 0 a 10 msec circa.**
- **transfer time:** il tempo che corrisponde all'abbassamento della testina sul disco, all'acquisizione dei dati e quindi al sollevamento della testina. Ovvero è il tempo necessario all'acquisizione vera e propria del dato che è stato raggiunto. (quindi attivazione e disattivazione delle operazioni di lettura). Durante questa operazione il disco continua girare quindi tipicamente un'operazione di questo tipo porta a leggere migliaia di byte e questo corrisponde all'unità minima di lettura ovvero la **"pagina di disco"** (è quella parte di superficie di disco a cui si accede atomicamente (in un'unica lettura non si può leggere né più né meno di una pagina)). **Transfer rate da 1msec per 4KB page.**

Note:

- In memoria centrale, il random access ha un costo indipendente dalla sua posizione (non ha quindi il concetto di "continuità fisica").
- Il settore contiene più pagine, una per traccia, e corrisponde a una porzione di disco caratterizzata dal fatto che gli spostamenti longitudinali del braccio dal centro verso l'esterno (e viceversa) vanno a leggere le pagine che stanno sullo stesso settore. Dal momento che le operazioni di lettura vengono fatte su una specifica traccia, si specifica "l'indirizzo" ovvero l'informazione relativa al settore e alla traccia in cui il dato interessato è posizionato. L'informazione sul settore dice in che momento della rotazione bisogna abbassare la testina per iniziare la lettura mentre l'info sulla traccia dice a quale distanza dal centro del disco si trova l'informazione di interesse.
- La rotazione del disco viene a velocità costante.

Concetto logico "next block": per "block" si intende la parte leggibile. Si intende dire che la situazione ideale sarebbe quella di riuscire a memorizzare le pagine, che verranno lette consecutive, in un ordinamento fisico che ricalchi il concetto di "next page" (o "next block"). Idealmente se si sa che dopo una pagina ne servirà un'altra, questa potrebbe essere adiacente alla precedente e sulla stessa traccia perché in questo modo la lettura della pagina avviene senza l'operazione di seek fra le due. Subito dopo si vorrebbe la prossima pagina sullo stesso cilindro (se la lettura avviene su un altro piatto ma sullo stesso cilindro, comunque l'operazione avverrebbe senza spostare il braccio). Ancora dopo si va a preferire un cilindro adiacente. Questo concetto corrisponde a un'organizzazione logica dei blocchi del disco che ottimizza le letture sequenziali sul disco. È quindi un concetto logico, non fisico, che dice quali saranno le letture successive (prevedibilità) verosimilmente meno costose data la posizione si sta occupando in questo momento (è ciò che minimizzerà il tempo di seek).

Sfruttano pesantemente il concetto di “next” risulta anche più semplice effettuare le operazioni di **prefetching (giocare di anticipo e portare in memoria centrale le pagine che stanno passando sotto la testina prima ancora che vengono richieste ma sapendo che sono state memorizzate secondo un ordine logico che verosimilmente corrisponderanno all’ordine di uso -> vantaggi in termini di tempo e throughput)** delle pagine perché nel momento in cui si legge, si passa fra pagine adiacenti e si sa che servirà nel breve.

‘Next’ block concept:

- blocks on same track, followed by
- blocks on same cylinder, followed by
- blocks on adjacent cylinder

Strutture di file e pagine: i file sono visti come una sequenza di pagine.

- **RID:** record ID, è l’identificatore del record (ovvero della tupla/riga della relazione)
 - Esistono sostanzialmente 3 tipi di file:
 - o **Heap (random order) files:** corrispondono a file in cui l’informazione è organizzata in modo NON ordinato, non c’è alcun criterio di ordinamento. Tipicamente l’accesso avviene scansionando il file e restituendo tutti i record. Contengono una “**collezione non ordinata di record**”. Dal punto di vista della memorizzazione sono quelli più semplici perché non ci sono vincoli di organizzazione ma dal punto di vista dell’utilizzo è chiaro che non avendo alcun vincolo non si ha alcuna proprietà da sfruttare. Ha un costo basso di gestione ma da poco supporto alle operazioni di ricerca. Tipicamente gli accessi agli heap file avvengono attraverso l’indicizzazione permettendo quindi di scansionare aree inferiori del file, senza di questi ogni ricerca può potenzialmente navigare l’intero file.
- Quando si crea un file di tipo head bisogna fare due cose:
- vedere come i record non ordinati si organizzano all’interno di tutta la pagina e poi bisogna (**organizzazione intra-page**, ovvero all’interno della pagina)
 - vedere come organizzare la pagine all’interno del file (**organizzazione inter-page**, ovvero tra pagine diverse per andare a costituire il file nel suo complesso)
- Il file sarà una collezione di pagine ciascuna delle quali è una collezione di record.
- È possibile definire un indice per criterio.
- o **Sorted files:** è consigliato se bisogna restituire i record in un determinato ordine (o se c’è bisogno solo di uno specifico range di record). Tipicamente soddisfano due tipi di proprietà:
 - **ordinamento rispetto a uno specifico criterio:** ordinati rispetto a qualche attributo (o combinazione di attributi) in maniera ad esempio crescente/decrescente. Ciò che è chiaro è che ogni file ordinato, è ordinato rispetto 1 criterio. Per ogni “relazione” salvata sul file si può avere al più 1 versione/memorizzazione “sorted” perché avere più versioni “sorted” vorrebbe avere più repliche di memorizzazione dello stesso file con tutti i problemi di allineamento.
 - **strategia di memorizzazione:** tipicamente quando si dichiara un file “sorted” rispetto a uno specifico criterio, le pagine consecutive del file (ordinato rispetto al quel criterio) sono memorizzate su disco in modo consecutivo (ovvero rispetto al principio del “next”) in modo da favorire la scansione sequenziale del file rispetto a un criterio di ordinamento. L’ordinamento porta benefici in fase di interrogazione quindi con un basso costo di lettura di potranno avere tutte le informazioni richieste.
 - o **Index files:** file di indice di cui gli Hash file costituiscono un caso particolare. Permette la ricerca per “search key”. L’indicizzazione basata su hash ha monta efficacia nel caso

dell'uguaglianza ma non nelle query di range perché non effettua un ordinamento ma facilita solo il raggiungimento.

- **Hash files**

Nota: tra gli heap files e i sorted files c'è un trade off in termini di semplicità di gestione della struttura piuttosto che facilità di valutazione delle query. Ad esempio, nei casi di query di range, con gli heap si avrà una scansione completa del file.

Strutture ad indice: sono delle strutture in cui si memorizza opportunamente delle coppie **search_key/record_address**. Per ogni chiave di ricerca, si associa l'indirizzo del record sul file in cui è presente quella chiave. La struttura a indice più popolare è il B-Tree (albero binario di ricerca).

Record a lunghezza fissa: sono i record per cui si sa fin dall'inizio quanti byte ciascun campo occuperà. Quindi la relazione (ovvero un record) è composto da un determinato di campi con dimensioni specifiche. Questo tipo di organizzazione permette un accesso immediato al campo interessato. Si ha vantaggi in termini di prevedibilità a discapito della rigidità dello schema, se per prudenza si impostano campi troppo ampi verrà sprecato dello spazio (anche se i campi saranno vuoti) -> un'alternativa sono i record a lunghezza variabile.

Il fatto che la lunghezza sia fissa semplifica l'organizzazione dei record nella pagina. Se N sono i record da usare, li si utilizzano l'uno dopo l'altro senza spazi intermedi (dopo il primo record c'è direttamente il secondo).

Page formats: Fixed length records

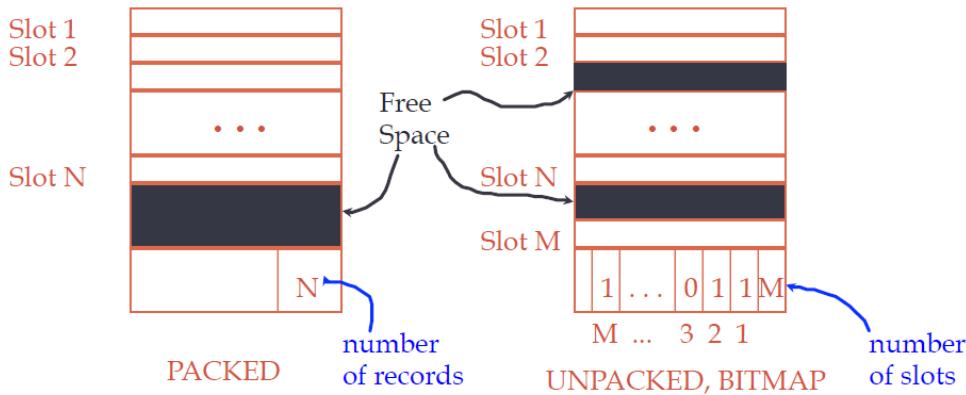
Due tipi di strategie: **packed** e **unpacked**.

Nota: cambiando un puntatore bisogna cambiare i successivi (update, c'è quindi un costo di gestione degli indici), questo ad esempio può avvenire nel caso di cancellazione.

RID (Record id) = <page id, slot #>

Gli slot sono tutti di pari lunghezza. Si parla di file Heap al momento e quindi ogni nuovo elemento può essere salvato nel primo spazio libero.

Per effetto delle cancellazioni effettuate, si vanno a generare degli spazi liberi che vanno a frammentare la pagina. Questi casi vengono gestiti con una tabella che ha una sequenza di bit che, per ogni possibile slot delle pagine, associa alla pagina stessa un bit che ha valore 1/0 a seconda se lo slot corrispondente possiede una pagina attiva o cancellata. Se gli slot nella tabella sono in ordine crescente o decrescente, dipende dagli algoritmi utilizzati (l'importante è che siano ordinati), quando si ragiona in binario tipicamente di legge da dx a sx ma la logica di lettura è appunto tipicamente correlata alla logica dell'operatore che consente di estrarre i bit (ovvero non è una logica architetturale del DB ma dipende dall'implementazione).



Record a lunghezza variabile: non definisce a priori la lunghezza del record ma vengono personalizzate le dimensioni dei campi dei record. Due alternative:

- **Field count:** è richiesto un carattere speciale come separatore del campo (non utilizzato all'interno dei campi). Il vantaggio è che non viene sprecato dello spazio per effetto delle sovrastime dei campi, fra gli svantaggi si trova l'impossibilità di accedere direttamente al campo (perché per arrivare al quarto campo bisogna iniziare a contare dal primo) e inoltre (tipicamente) la rappresentazione dei caratteri è fatta in modo da favorire i caratteri più utilizzati dando dimensioni più piccole ai caratteri più usati e più lunghe a quelli meno usati quindi si corre il rischio che la codifica del separatore sia molto lunga (lo spazio risparmiato lo si potrebbe impiegare con i separatori).
- **Array of field offsets:** è un approccio organizzativo diverso che si basa su un approccio a organizzativo a directory. La prima parte del record è dedicata a contenere un insieme di puntatori che puntano all'inizio di ciascuno dei campi (e l'ultimo puntatore invece punterà alla fine). L'array di offset sarà grande $n_fields+1$. Lo spazio dedicato alla directory è solitamente inferiore rispetto al primo approccio organizzativo per **field_count** (i puntatori occupano meno spazio rispetto ai caratteri speciali). Lo svantaggio è che dopo che si stabilisce l'allocazione del puntatore, si stabilisce anche il valore massimo che può assumere (quindi è una rappresentazione variabile ma con un **bound** dato dall'indirizzo massimo rappresentabile dati i puntatori iniziali).

Nota: guardare le slides sui record.

Page formats: Variable length records

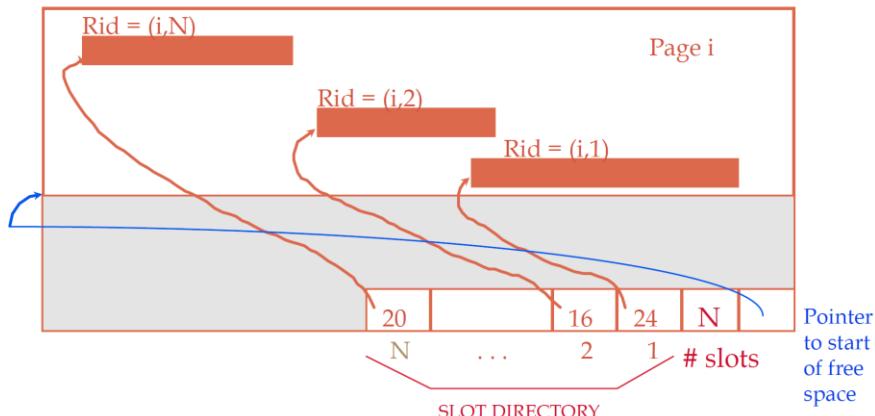
La gestione dei record con lunghezza variabile aggiunge, rispetto a quelli con lunghezza fissa, la complicazione del fatto che non si hanno record di pari lunghezza. Non si può dare per scontato che ciascun record che arrivi in scrittura possa esser scritto nello spazio di un record cancellato.

Si parla nell'ottica dei file heap, qualsiasi inserimento può essere fatto in qualsiasi spazio libero purché il record da inserire nello spazio libero ci stia (ovvero purché lo spazio libero sia sufficientemente grande per contenere l'oggetto che si vuole memorizzare). Strategia basata sul concetto di **directory** che contiene il puntatore al punto in cui inizia lo spazio libero oltre ai puntatori all'inizio dei record che sono fisicamente presenti nella pagina.

RID (Record id) = <page id, slot #>

Nota:

- Si può presentare il problema della frammentazione per cui si creano degli spazi vuoti troppo piccoli. Una soluzione è il compattamento ma il compattamento porta il problema dell'eventuale presenza di indici che puntano ai record (quindi bisogna aggiornare tutti gli indici definiti sui record dei file).
- Il compattamento NON è costosa perché avviene in memoria centrale. Evita il costo dell'aggiornamento degli indici e si risolve la frammentazione.



To support record level operations, we must:

- keep track of the pages in a file
- keep track of free space on pages
- keep track of the records on a page

There are many alternatives for keeping track of this.

Un'alternativa potrebbe essere avere un puntatore da uno slot al precedente e al successivo ma si avrebbero i limiti dell'accesso sequenziale (caso migliore nel primo slot, peggiore nell'ultimo e medio al centro del file). Uno degli aspetti negativi, di questa strategia semplicistica, è che a livello globale non si hanno informazioni su dove c'è lo spazio libero.

Un'altra soluzione per risolvere questo problema potrebbe essere gestire le aggiunte delle nuove pagine in questo modo: ogni volta si va all'ultima pagina, se c'è spazio lì bene se no altrimenti se ne crea una nuova. Ha il vantaggio di avere le operazioni di inserimento meno costose ma ha lo svantaggio di rendere potenzialmente più numerose le pagine dedicate all'intero file (oltre che allo spreco di spazio dovuto dalle eliminazioni). Avere tante pagine in un file vuol dire leggerne molte di più quando si legge/scandisce un file, questo è costoso. Il costo risparmiato nella scansione sequenziale durante gli inserimenti viene però impiegato in maniera maggiore durante le ricerche.

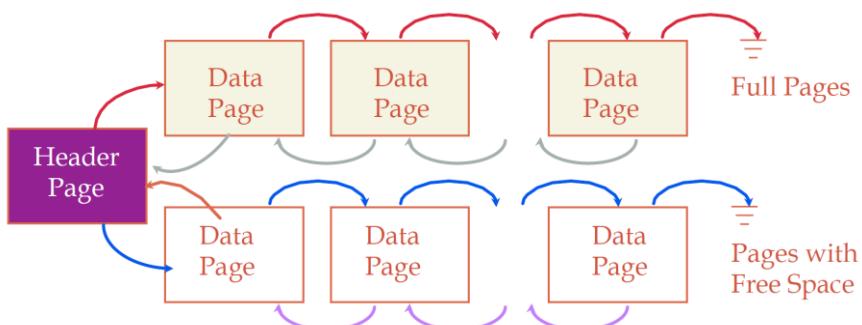
È necessario un trade-off. La soluzione di compromesso che si ottiene (di solito) fa riferimento all'utilizzo di due diverse categorie di pagine destinate a:

- una contenere pagine completamente piene e
- una destinata a contenere pagine con ancora dello spazio (parzialmente piene)

Quindi se la cancellazione avviene nelle pagine piene, queste vengono spostate nella lista delle pagine parzialmente piene. Se invece una pagina parzialmente piena si satura al seguito di inserimenti, allora viene spostata nell'altra lista. Ogni pagina contiene quindi due puntatori oltre ai dati. Questa soluzione è particolarmente vantaggiosa se le pagine sono a lunghezza fissa perché, in questo modo, durante gli

inserimenti non c'è bisogno di scandire tutta la lista per trovare uno spazio e verificare che il record ci stia (se solo c'è spazio in una pagina, allora lo si inserisce). Se i record sono a lunghezza variabile, non è detto che in una pagina ci sia sufficiente spazio per contenere un nuovo record quindi l'aggiunta comporterebbe una scansione sequenziale della lista.

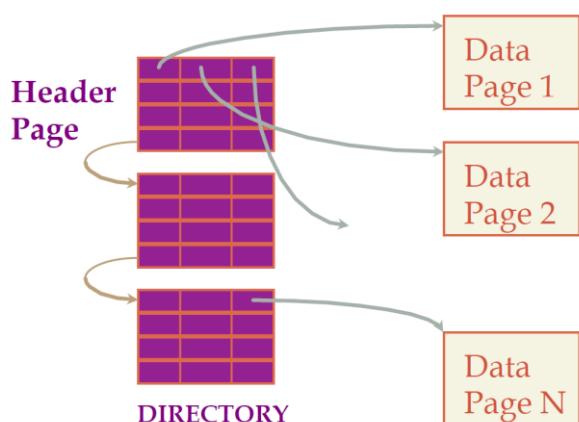
Heap File Implemented as a List



Per risolvere questo problema si può ricorrere al meccanismo della directory. Si vede il file come una sequenza di pagine a cui si accede attraverso una directory. La directory contiene delle coppie di informazioni che sono: puntatore alla pagina e l'informazione relativa allo spazio libero in quella pagina. Se i record sono a lunghezza variabile, questa strategia è potenzialmente vantaggiosa. È vero che comporta comunque una scansione sequenziale per cercare la pagina ma, data la natura dei dati, ciascuna pagina della directory contiene dati riguardanti un numero elevatissimo di pagine

($\text{dimensione_pagine}/(\text{dimensione_puntatore}+\text{dimensione_numero_checontienelospaziolibero})$).

È una lettura sequenziale però molto corta rispetto alla lettura di un file. La directory ha la funzione di tener conto dove c'è lo spazio libero. La pagina di directory è la pagina di memoria, non ha una grandezza fissa perché in quel caso limiterebbe la grandezza del file, questa directory è parte del file stesso.



Indexes: sono dei file che intervengono per accelerare gli accessi ai dati che sono contenuti nei file di tipo heap o sorted. Velocizza la selezione per campi chiave su cui sono stati costruiti gli indici. In assenza di indici,

in lettura viene effettuata l'intera scansione del file. I vantaggi dei sorted files è avere un ordinamento che, se affiancato agli indexes files, favorisce notevolmente le operazioni in lettura. Il vantaggio della memorizzazione ordinata e continua (ovvero le pagine relative allo stesso file sono una di seguito/attaccata all'altra) è il tempo per raggiungere le informazioni che in questo momento è logaritmico, esattamente come la ricerca binaria, al prezzo della gestione del mantenimento dell'ordinamento a fronte degli updates e anche al prezzo della consapevolezza che il criterio di ordinamento su un file è uno solo. Se è sorted rispetto ai cognomi, non è possibile che lo sia contemporaneamente sorted per data di nascita. Per ogni file c'è solo una versione sorted (ma ci possono naturalmente essere più al costo dello spazio). Se c'è un criterio di ordinamento, ce n'è uno solo. Hanno solo scopo di "snellire" le operazioni di accesso ai dati secondo le chiavi di ricerca.

Tempo: logaritmico (indici) vs lineare (no indici).

Due classi di indici: hanno proprietà e caratteristiche di diverse, nonché utilizzi diversi

- **Gerarchici:** il più utilizzato è il b+ tree. Sono molto adatti a rispondere a query di ricerca rispetto a una specifica chiave ma anche query di range (ovvero ad accesso di intervalli di valori chiave su cui l'indice è definito). L'organizzazione dei dati è tale per cui la struttura a indice rispetta l'ordinamento dei dati.
- **Hash:** basati sulle tabelle di hash. È possibile avere chiavi di ricerca molto vicine che invece vengono mappate su pagine dell'indice diverse e quindi l'utilizzo dell'indice, mediante tabella di hash, è molto efficace per l'accesso diretto alla singola chiave ma non è utilizzabile per rispondere in maniera efficace ed efficienti alle query di range.

Nota:

- per **chiave di ricerca** si intende il valore dell'attributi rispetto a cui si effettua la ricerca. "**chiave di ricerca**" è indipendente rispetto alla "**chiave principale della relazione**".
- **Chiave di una relazione:** è un sotto insieme di attributi della relazione che consente di identificare in modo univoco i record della relazione. Ogni relazione può avere più **chiavi candidate**, tra le diverse chiavi candidate se ne sceglie una come chiave principale. Esiste anche il concetto di **chiave minima**. Sono gli identificatori di riga delle tabelle.
- **Chiave di ricerca:** quando si effettua la ricerca di un record che su uno specifico attributo assume un certo valore, nel contesto della query la chiave di ricerca è l'attributo che deve assumere lo specifico valore (trovare gli studenti che si chiamano "mario", la chiave di ricerca sarebbe il nome in questo caso). È il valore specifico (o un range) per un attributo per cui si è interessati per rispondere a una query. Corrisponde all'attributo che si sta cercando per rispondere alla query.
- Si definiscono indici diversi per ciascuna chiave di ricerca a cui si è interessati, quali accessi (ai record) a cui si è interessati a renderli più efficienti.

Alternatives for Data Entry k^* in Index

- In a data entry k^* we can store:
 - Data record with key value k , or
 - $\langle k, \text{rid of data record with search key value } k \rangle$, or
 - $\langle k, \text{list of rids of data records with search key } k \rangle$
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value k .
 - Examples of indexing techniques: B+ trees, hash-based structures
 - Typically, index contains auxiliary information that directs searches to the desired data entries

Nota: i tre tipi evidenziati nella foto sono i tipi di informazioni che si possono memorizzare negli indici-

1. dati del record con la chiave.

Vantaggi: presuppone l'organizzazione ordinata di tutti i record di tutta la relazione. Siccome si può avere l'ordinamento della relazione rispetto a un criterio solo, il mantenimento dell'ordinamento rispetto all'intera sequenza di record dell'intera relazione lo si può fare per un indice solo.

Svantaggi: il fatto di avere l'intero record all'interno dell'indice comporta un indice pesante dal punto di vista dello spazio.

2. nel file si hanno le informazioni limitate alla chiave di ricerca su cui si sta costruendo l'indice (ex $\langle \text{cognome, indirizzo_record_con_cognome_ex_rossi} \rangle$). Una coppia $\langle k, \text{address} \rangle$ per ciascun record fisicamente presente nella relazione che si sta indicizzando. Per ogni chiave, eventualmente ripetuta, ha tanto corrispondenze della chiave con i corrispettivi RID (tante occorrenze della chiave quante sono le ripetizioni di quella chiave).

Vantaggio: essendo una relazione basata tutta sulle coppie $\langle k, \text{RID} \rangle$, queste informazioni hanno lunghezza fissa e quindi si possono gestire come i fixed record e semplifica in termini di stime (se è variabile è per via del campo chiave ma non per il numero). Non c'è da gestire la variabilità.

Svantaggio: tante occorrenze delle chiavi.

3. Simile al secondo ma invece che avere una copia per record, si ha un'associazione $\langle k, \text{list_addresses} \rangle$. Se si hanno N studenti che si chiamano "Rossi" di cognome, nel secondo caso si avranno N coppie con cognome Rossi, nel terzo invece solo una con chiave e lista di RID che corrispondono a questa chiave di ricerca. I record hanno lunghezza variabile.

Vantaggi: meno occorrenze delle chiavi.

Svantaggi: variabilità da gestire.

Range searches: sono le query/interrogazioni di range. Si ha un disco rigido diviso in pagine, si ha dei file memorizzati sulle pagine, si assume che un certo numero N di pagine (non necessariamente memorizzate in maniera continua) sono quelle che costituiscono il data file a cui si è interessati. L'idea è di creare una struttura dati ausiliaria grazie la quale si riesca a trovare l'elemento che si vuole raggiungere con la ricerca in modo diretto/guidato risparmiando il costo della scansione sequenziale (l'accesso in memoria secondaria ha già un costo alto). Il primo puntatore punta all'inizio del data file. Quindi si crea una struttura ausiliaria in cui ai diversi valori della chiave di ricerca si associa il puntatore al record che contiene il valore di chiave nel campo corrispondente.

Il vantaggio di avere questo file separato è facilitare gli accessi. I vantaggi della struttura ausiliaria:

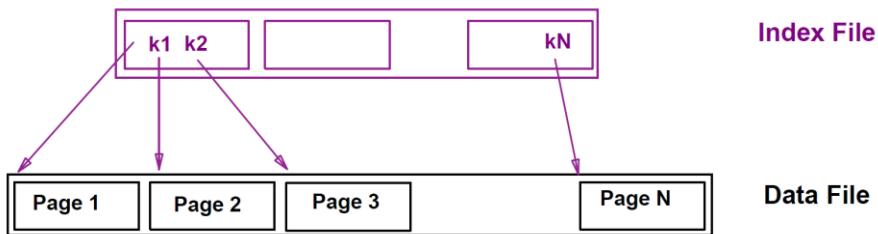
- è ordinata

- la si può memorizzare compatta (la coppia ordinato-compatto consente il tipo di ricerca binaria molto vantaggiosa)
- occupa meno spazio

Svantaggi:

- Spazio: per quanto è piccola, occupa della memoria in più
- Aggiornamento efficacie: gestione degli update in modo da mantenere gli update
- Salto in più per l'accesso: se si è interessati a dati che vanno oltre la chiave su cui si fa la ricerca (ad esempio: trova l'indirizzo degli studenti il cui cognome comincia per "n" -> tramite l'indice si trova lo studente ma poi bisogna fare il salto in più al record dello studente la quale chiave corrisponde a quella che si vuole trovare)
- Il file costruito come struttura intermedia è ancora potenzialmente lungo, cioè se si suppone che la chiave di ricerca sia circa 1/10 (come ampiezza) rispetto al record globale, se si ha un record che occupa 1mln di pagine, questo ne occuperà $1\text{mln}/10 = 100\text{k}$ (è ancora grande perché si ha un rapporto 1:1 tra chiave e record). Quindi è consigliabile salire di livello: non verranno indicizzate tutte le chiavi ma solo degli intervalli (valori fra Kn e Km), è possibile perché il file è ordinato e lo si può vedere come una concatenazione di intervalli.

NOTA: l'ordinamento è un criterio necessario nella struttura ausiliaria.



Note:

- Il file aggiuntivo è ordinato quindi se si vogliono più ordinamenti è sufficiente avere più strutture di appoggio con ordinamenti differenti.
- È possibile associare alla chiave di ricerca il page_id invece del RID. Il page_id è parte del RID. Il RID è tipicamente costituito dalla coppia <page_id, displacement>, se si associa solo il page_id bisognerebbe poi andarsi a cercare il record all'interno della pagina. Se si è interessati a portare in memoria centrale solo la pagina è sufficiente il page_id ma in questo caso bisogna cercare all'interno della pagina, in memoria centrale, la chiave di lettura mentre in il RID si raggiunge direttamente il record grazie al displacement.
- La struttura ausiliaria potrebbe essere memorizzata in maniera continua e quindi potrebbe essere trattata come un sorted file a differenza del data file. Quindi sarebbe possibile effettuare una ricerca di tipo binario nel sorted file

Organizzazione unclustered (non clusterizzata): questo tipo di organizzazione, in cui non c'è alcun tipo di ordinamento imposto sul data file e non c'è alcun tipo di contiguità richiesta sulla memorizzazione del data file, dove il livello delle foglie dell'indice è quello che costruisce l'ordinamento sui dati presenti nella relazione stessa. Non c'è alcun requisito di ordinamento dei record logicamente vicini nelle pagine dei dati, sono

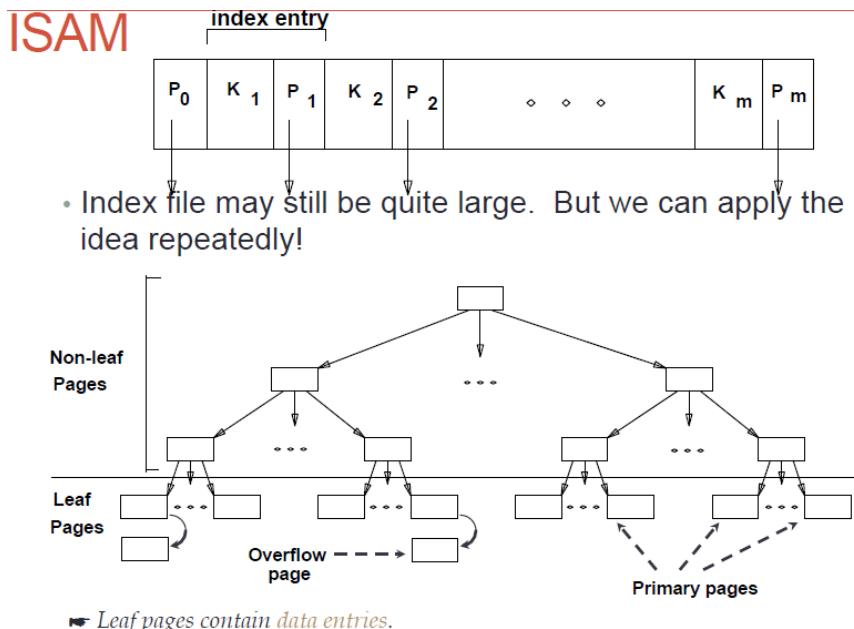
logicamente e fisicamente vicine le chiavi nelle foglie dell'indice e questo ordinamento non corrisponde necessariamente all'ordinamento nelle pagine nel disco (attenzione perché il lucido è fuorviante).

ISAM (Indexed sequential access method): indicizzando a intervalli si costruisce una struttura gerarchica che consentirà un accesso efficace ai dati. Nel momento in cui si costruisce questo indice, la ricerca si effettua come su un albero. È un metodo per l'accesso sequenziale indicizzato. Accesso agli indici con costo log2. Il file di dati è organizzato in maniera contigua. Si conclude che un dato non c'è al seguito dell'intera scansione di un cammino.

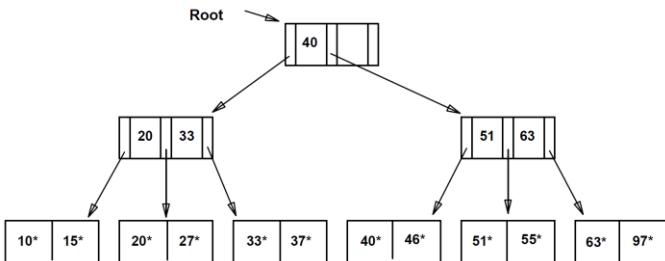
Se le foglie sono troppe e l'accesso sequenziale può essere pensato, si sale al livello superiore che indica il range e via dicendo si sale in maniera ricorsiva fino a raggiungere la radice (più si sale e più i range saranno ampi). Questo tipo di struttura si chiama **ISAM**.

Questo tipo di struttura presuppone che tutti i nodi siano riempiti completamente. Questa particolare struttura presuppone che non si gestisca, in modo integrato con la struttura stessa, l'eventuale dato che arriva a fronte di inserimenti. Ovvero, al seguito di un'aggiunta, per bilanciare l'albero si identifica quella che sarebbe la pagina (**primary page**) in cui il dato dovrebbe essere inserito e se non ci sta si genera una **pagina di overflow (overflow pages)** che estende quella precedentemente identificata (ex con il numero 47 si identifica la quarta pagina che però viene estesa con una di overflow perché non ci sta).

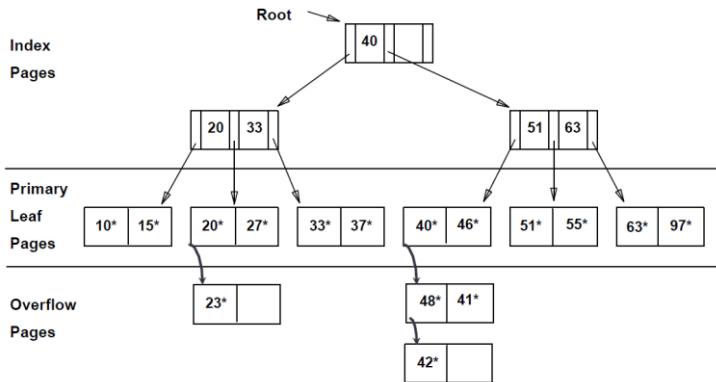
L'overflow ha un comportamento di deterioramento della struttura dell'indice, per questo motivo questo tipo di indice è molto efficace in realtà molto statiche (con pochi inserimenti e poche cancellazioni). In nature statiche è un indice semplice da gestire ed efficace, i casi con molta dinamicità vengono gestiti diversamente (ad esempio con i B+ Tree che hanno una gestione della dinamicità che permettono di fronteggiare meglio le realtà con tanti inserimenti e tante cancellazioni).



Esempio di ISAM Tree: naturalmente le pagine con solo due valori non sono realistiche. Questo indice dice nelle foglie quali sono tutte le chiavi presenti per questo specifico attributo e che ci sono 6 pagine (6 pagine sono ancora tante perché a livello realistico conterebbero molti più valori).



After Inserting 23*, 48*, 41*, 42* ...



L'accesso ai dati su questa struttura, ossia quante pagine si devono portare in memoria principale e quante operazioni I/O per raggiungere l'operazione che serve, bisogna farne tante quanto la profondità dell'albero che contiene le sole chiavi di ricerca +1 che è il dato che contiene effettivamente il dato interessato.

- Meno è profondo l'albero, più è efficiente il suo utilizzo. Il fattore di ramificazione e il bilanciamento dell'albero influiscono sull'efficienza dell'albero.
- Ciascun nodo interno ha la dimensione di una pagina di memoria (è una dimensione fissa).
- I puntatori hanno la dimensione degli indirizzi di pagine di memoria (sono tipici dell'architettura).
- La dimensione della chiave di ricerca è un altro elemento che si può usare per fare leva
- Avere dimensioni fosse aiuta in termini di efficienza
- La profondità di un albero è data dal log di base fanout del numero degli elementi da indicizzare. Ad ogni livello, il numero di nodi che si vanno a trovare vengono "scalati" (la "scalatura" dipende da questo fattore). Il numero di nodi che si ottengo, si ottengono dividendo quelli del livello più basso * fatto di fanout. Il fattore di ramificazione massimo che si può avere (max fanout) è dato dal rapporto tra dim_pagina_disco e la dim_informazione_associata alle singole chiavi di ricerca (ovvero la dimensione della coppia key-RID, quindi chiave-puntatore per la rappresentazione del dato stesso).
- Il costo è dato dalla profondità +1. Al posto di +1 è potenzialmente +N se N sono gli elementi aggiunti a livello di foglie.
- Ogni accesso al dato comporta il caricamento dell'intera pagina in memoria principale.
- Sia gli inserimenti che le cancellazioni non vanno a modificare la struttura interna dell'indice. Gli inserimenti inseriscono nelle foglie, quindi nei file di dati, se c'è spazio. Le cancellazioni cancellano dal file di dati ma la corrispondente chiave, se presente nell'indice, ci resta. Il file di indice è statico, può cambiare solo se si nota che la struttura sta degenerando e il costo sta diventando eccessivo perché magari ci sono troppe pagine di overflow.
- il file è sorted (ordinato), contiguor, ecc. L'**implicazione** è per ogni relazione ci può essere al più 1 ISAM, ogni relazione può essere ordinata rispetto ad un unico criterio di ordinamento (questo vincolo viene meno quando si parla di B+ Tree).

Nota:

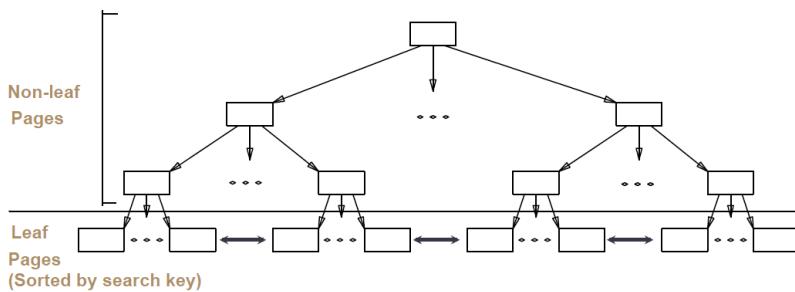
- **fanout:** fattore di ramificazione è definito come num_puntatori_presenti_nel_nodo. Il fattore di ramificazione corrisponde al numero di puntatori presenti nel nodo (nel caso sopra sono 3, ogni nodo contiene 3 chiavi e quindi 3 figli).
- L'asterisco negli indici dei lucidi ricorda che si sta parlando di record.

Svantaggi:

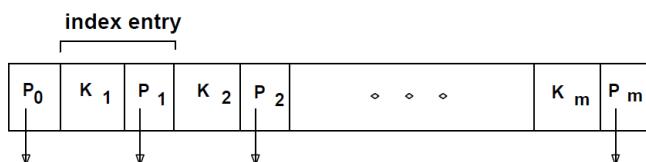
- è una struttura molto efficiente solo in realtà statiche. L'aggiunta di altri elementi al seguito della saturazione della struttura, porta alla generazione di pagine di overflow che peggiorano l'efficienza.

B+ Tree: più utilizzato rispetto a ISAM perché più realistico. Rispetto a ISAM, continua ad avere i dati nelle foglie ma non sono più memorizzati in modo ordinato e contiguo allora si usano i puntatori (avanti e indietro) per passare da una foglia all'altra, viene quindi meno il vincolo di poter avere un unico criterio di indicizzazione. È possibile fare al più 1 indicizzazione di tipo ISAM per ciascuna relazione, con i B+ Tree questo vincolo viene meno.

Per effettuare le query di range, nel momento in cui viene meno la continuità sono necessari i puntatori.

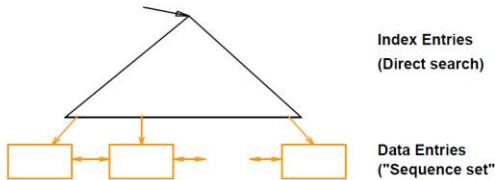


- ❖ Leaf pages contain *data entries*, and are chained (prev & next)
- ❖ Non-leaf pages have *index entries*; only used to direct searches:



B+ Tree: Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. ($F = \text{fanout}$, $N = \# \text{ leaf pages}$)
- Minimum 50% occupancy (except for root). Each node contains $d \leq m \leq 2d$ entries. The parameter d is called the *order* of the tree.
- Supports equality and range-searches efficiently.



Note:

- Le foglie contengono i dati, sono concatenate e, come nell'ISAM, i nodi interni contengono esclusivamente la chiave d'accesso su cui si sta definendo l'indice e i puntatori.
- Nelle foglie c'è il dato completo.
- M copie key-pointer, in fanout in questo caso è $M+1$.
- I nodi interni non sono mantenuti in modo statico e non completamente riempiti. La struttura dell'albero rimane bilanciata ma la struttura dell'albero viene aggiornata a fronte di aggiornamento che ne richiedono necessario, ovvero non ci sarà l'overflow. Si tengono i nodi interni riempiti almeno al 50% e al più al 100%. L'occupazione media delle pagine interne è del 70%, questo comporta un minore sfruttamento dello spazio (rispetto all'ISAM). Minor riempimento comporta minor fanout e quindi una profondità dell'albero maggiore MA, con questa organizzazione, permette di passare dalla radice alle foglie con l'attraversamento dell'albero e una volta arrivati alla foglia si ha il dato (no pagine di overflow). Vengono usati algoritmi di aggiornamento dell'indice.
- Si cerca di avere l'albero sufficientemente alto in modo da aumentare il fanout in modo da rischiare il meno possibile di trovarsi in un noto interno e avere poco margine per ridirigere il traffico e raggiungere l'informazione realmente ricercata.
- È un albero bilanciato con complessità logaritmica del fanout con la garanzia che ciascun nodo sia riempito almeno al 50% e max 100%. Con un'occupazione di questo genere, il numero di chiavi varia tra d (che è il numero di chiavi che si possono ottenere con un'occupazione al 50%) e $2d$ (che è il numero di chiavi con l'occupazione al 100%). Il **parametro d** viene chiamato **ordine dell'albero**.
- **Ordine del B-albero:** è il numero minimo di chiavi contenute in un singolo nodo. Questo ha impatto sul fattore di fanout che è +1. Ad esempio, grado 2 significa dimensione max pari a 4.
- Come l'ISAM supporta query di uguaglianza e range in maniera molto efficiente.
- Essendo la dimensione del nodo pari alla pagina di memoria, gli elementi che stanno in un nodo stanno nell'ordine delle centinaia di migliaia in base a quanto è grande la chiave indicizzata.
- La ragione per cui si tengono tanti spazi vuoti è perché in quello modo si riducono le operazioni di ristrutturazione che sono molto costose (a discapito dello spazio).
- L'albero non può risultare MAI sbilanciato perché quando si colmano i nodi, gli splitting hanno impatti sui genitori andando quindi a mantenere il bilanciamento.
- Solo la radice può avere un riempimento <50%.
- I B+ Tree sono quelli che nei nodi interni memorizzano solo le chiavi, nei B-Tree tradizionali i nodi interni non contengono solo le chiavi ma anche il dato stesso.
- Il fanout è legato al numero massimo di coppie key-puntatore che si possono memorizzare all'interno della pagina.

- Il B+ tree non presuppone che il file indicizzato sia già ordinato di suo quindi il file arriva disordinato ed è compito dell'indice renderlo ordinato a livello delle foglie.

Prefix key compression: è una soluzione di compromesso nel caso in cui si ha la stretta necessità di avere chiavi molto grandi (ad esempio un indirizzo) poiché la natura del dominio giustifica tante interrogazioni su quella chiave. Si potrebbe avere una funzione di hash per associare un valore numerico alle chiavi ma lo svantaggio è che bisognerebbe applicare la funzione di hash ogni volta, per questo si è passati alla tecnica della **compressione del prefisso** (è una tecnica più efficace). È una tecnica prevede che, qualora le stringhe siano troppo lunghe, anziché inserire a livello di chiave l'intera stringa si può inserire a livello di chiave soltanto la porzione di stringhe che serve per **discriminare** e quindi dirigere il traffico (quindi ciò che serve per poter capire quale puntatore seguire). Queste compressioni potrebbero avere come effetto collaterale l'aumento del fanout, quindi avere casi in cui le chiavi (nodi interni) vengono rappresentati non nel suo complesso ma "quanto basta". Le implicazioni di questa scelta sono:

- Il fatto che il prefisso sia discriminante (e quindi sufficiente per rappresentare le chiavi) è vero in una determinata soluzione ma non sarebbe vero se poi si aggiunge, o c'è, qualche altra chiavi non sufficientemente discriminata. Per discriminare non basta guardare quindi solo il nodo ma anche la chiave più vicina presente nel sotto albero adiacente. (questo è un effetto)
- Costo. La sotto stringa discriminante, a seconda dei dati che si stanno gestendo, anche lei può diventare parecchio grande quindi per gestire un caso di questo genere bisogna passare ad un'architettura che permetta la rappresentazione dei record dell'indice mediante record variabili (ciò che invece non era strettamente necessario con i record a dimensione fissa). Il vantaggio dell'appiattimento dell'albero lo si paga al costo della gestione della variabilità del record a livello di singolo nodo (questa fa saltare l'idea di avere un determinato grado per i nodi, nodi con stringhe più lunghe e altri più corti). Inoltre, le procedure di inserimento e cancellazione sono pesanti poiché bisogna assicurarsi che le sotto stringhe delle chiavi continuino a discriminare pena l'aggiornamento anche delle chiavi (il back-tracking vanificherebbe l'utilizzo dell'indice).

Bulk loading of a B+ Tree: ciò che tipicamente si effettua quando si crea da zero un indice su una relazione, è l'operazione di **bulk loading**. È un'operazione che carica in maniera completa, compatta ed efficiente le informazioni all'interno dell'albero. Le idee alla base di questa organizzazione efficiente per l'inizializzazione:

- obiettivo: dato un file che contiene un certo insieme di data entry, organizzare le data entry in un B+ Tree che in quanto tale abbia le foglie che contengano le foglie che contengano dati ordinati e farlo nella maniera più efficiente possibile. È un'operazione che richiama l'inizializzazione dell'ISAM, ovvero si parte da un file ordinato (in questo caso lo si sta facendo partendo da una relazione non ordinata). Se ci si può permettere di ordinare la relazione completamente (ovvero quando il file non è già stato ordinato rispetto altri criteri da un altro B+ Tree o ISAM), si ordinano completamente le pagine della relazione su cui bisogna costruire l'indice in modo che le pagine ordinate rappresentino il livello delle foglie dell'indice stesso. Se non ci si può permettere di farlo (ad esempio perché c'è già un ISAM su un altro criterio) ciò che si ordina, per costruire il livello delle foglie dell'indice che si sta costruendo, è solo insieme delle chiavi su cui si sta costruendo l'indice quindi si fa una copia delle chiavi-puntatori e si effettua un ordinamento rispetto a loro. È una tecnica costosa ma ci sono metodi efficienti.

Il bulk loading ha la seguente proprietà: i record ordinati vengono inseriti (una pagina alla volta) nella struttura e ciascun inserimento ha impatto solo sul sottoalbero più a destra che ho già inserito nella struttura che sta crescendo. Quindi prima si effettua l'ordine delle foglie e poi si costruiscono i nodi interni ma sempre

con effetto solo a dx con costo contenuto (perché le operazioni avvengono solo a dx, l'ordinamento dei dati di partenze garantisce questo genere di comportamento). Con il bulk loading si riesce a mantenere i fattori di riempimento iniziali scelti dal progettista. Le foglie possono essere memorizzate in maniera sequenziale o mediante link in funzione del fatto se sono già state memorizzate sotto un certo ordine o meno (ovvero se è già stato utilizzato un'altra indicizzazione che prevedere il sorted index). Quando i record sono a lunghezza variabile è difficile tenere pagine contigue, la garanzia sulla cardinalità delle chiavi non si può avere.

Classificazione degli indici:

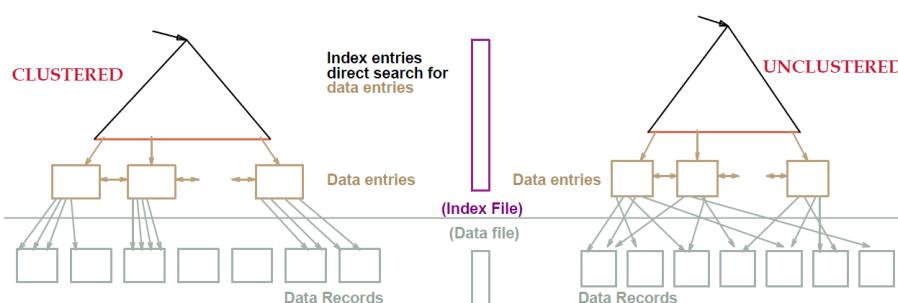
- **Indice primario:** sulla chiave primaria, tutte le relazioni hanno un indice sulla chiave principale
- **indice secondario:** indice su altre chiavi

Clustered vs Unclustered index: clustered è ad esempio la struttura ISAM che impongono l'ordinamento mentre le strutture unclustered sono quelle che non impongono l'ordinamento.

- un indice è clustered se l'ordinamento presente a livello di foglie dell'index file è coerente con l'ordinamento dei record dei dati nel file dei dati. Vuol dire che il file, su cui l'indice è costruito, è ordinato oppure "quasi ordinato" (ad esempio il file dati ISAM con pagine over flow, quindi con qualche eccezione locale nell'ordinamento locale). Tipicamente succede che i puntatori che si incontrano scandendo le foglie ordinatamente (che per definizione sono ordinate perché la costruzione è fatta sull'ordinamento) sono a loro volta anche loro (i puntatori) ordinati.
- nel caso di indice unclusterd, se si vuole effettuare una scansione sequenziale e ordinata del file la si può fare a partire dall'indice ma non si ha la garanzia di leggere ciascuna pagina del file di dati una volta sola (cosa che invece accade con i clustered perché sono ordinati) proprio perché procedendo ordinatamente è possibile che si debba tornare alla prima pagina dopo aver raggiunto la quindicesima (ad esempio). Questo perché magari l'ordinamento del file di partenza era secondo un'altra relazione (come il cognome, l'indirizzo, il cellulare ecc). NON c'è coerenza tra l'ordinamento delle chiavi e delle foglie.

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.

- To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
- Overflow pages may be needed for inserts. (Thus, order of data recs is 'close to', but not identical to, the sort order.)

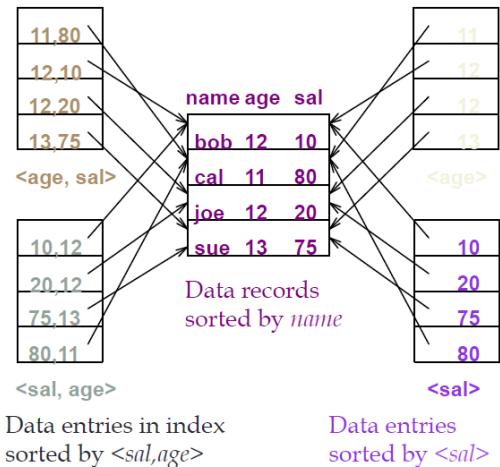


Indexes with Composite Search Keys: composite search è una ricerca su una combinazione di campi (equality query, range query). Sono numerosissime le query che riguardano più di un attributo. Si parla quindi

di "chiavi di ricerca composite", ovvero costituite da più di un attributo. Condizione di ricerca costituita da più criteri. La scansione sequenziale degli indici potrebbe comportare tanti salti (tante scansioni sequenziali quante possono).

L'indice composito è molto comodo per query che sono di uguaglianza sul primo degli attributi della coppia e se c'è un AND con i successivi campi, allora si è sicuri che quei campi soddisfano la richiesta, in caso contrario vanifica.

Examples of composite key indexes using lexicographic order.



Hash-based- indexes: è una struttura ad indice alternativa. Funzionano bene solo per le query di uguaglianza (quindi si possono usare solo in questo caso). L'indice è una collezione di **buckets**.

- **Bucket:** primary page +0 o più overflow pages
- i buckets contengono le entries

La funzione di hash è una funzione che dato in input un dato oggetto, lo mappa in un numero intero che corrisponde (che indica) al contenitore/bucket/pagina_memoria che contiene il dato.

Hashing function h: $h(r) = \text{bucket in which (data entry for) record } r \text{ belongs}$. h looks at the search key fields of r . No need for "index entries" in this scheme.

Nota: l'hash della chiave punta all'area di memoria che contiene l'informazione. Altre tecniche indicizzano directory di chiavi, fato l'hashing si cerca la chiave e se c'è questa punterà alla parte del disco che contiene le informazioni.

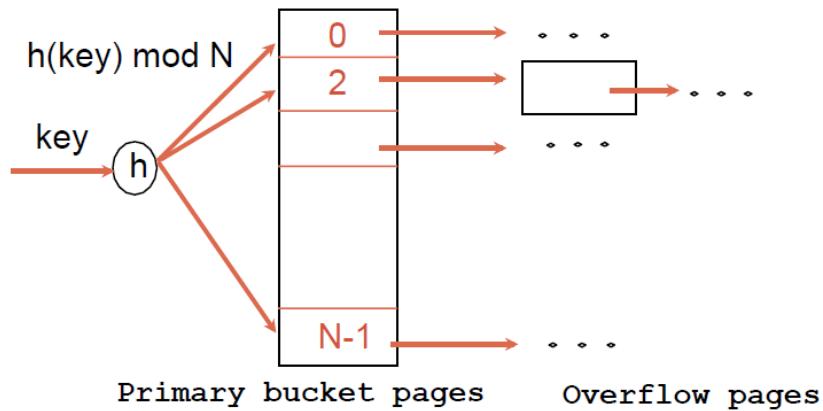
due metodi di fare hash:

- hash(ugo) = 1287 #hash direttamente sui dati che dicono che il dato si trova nella pagina 1287. Il vantaggio è che nel momento in cui viene applicata la funzione di hash, viene restituito l'indirizzo che è subito accessibile. Lo svantaggio è che la pagina in questo caso contiene poche chiavi (?).
- hash per trovare l'indice per trovare la directory (...). L'hash restituisce un numero che identifica una pagina nel disco in cui si troverà la coppia key-pagina_effettiva del disco in cui c'è la chiave.

Static hashing: le pagine primarie sono statiche, allocate sequenzialmente e mai deallocate, se necessario vengono utilizzate le pagine di overflow. In questo caso il file di hashing potrebbe essere troppo grande oltre che sottostimato. Le pagine non sfruttate hanno un impatto sulle performance, sono comunque da portare in memoria principale. Per questo motivo non è un'ipotesi molto usata a meno che le collisioni non siano un

problema (ad esempio se le collisioni sono ben distribuite che portano a un bilanciamento di carico sulla diverse chiavi risultati).

$h(k) \bmod M$ = bucket to which data entry with key k belongs. ($M = \#$ of buckets). La funzione modulo N restituisce sempre una valore da 0 a $M-1$, quindi restituira sempre un bucket compreso nell'intervallo.



Nota: se si prendono in considerazione i cognomi è molto probabile che ci siano più Rossi che Sapino, quindi è molto probabile che i Rossi non ci stiano nella pagina e quindi si aggiungono le pagine di overflow (e si generano quindi delle catene di overflow). Questo accade perché le pagine sono statiche.

Nota: 0, 2, ..., $N-1$ sono indirizzi di pagine. Quando ci sono delle collisioni in realtà i dati finiscono nella stessa pagina, quando la pagina è piena di usano le pagine di overflow ma se ce ne sono troppe e l'albero inizia a essere non bilanciato vuol dire che si stanno iniziando a perdere le performance. Il costo di uso di questa struttura aumenta all'aumentare delle catene di overflow perché ammettono gli accessi a disco (se una catena di overflow è lunga 20, vuol dire che l'accesso ai bucket è almeno 20 per poter accedere alle pagine). Bisogna trovare una via di mezzo fra una buona ottimizzazione di bucket (pagine riempite bene) e non avere catene di overflow troppo lunghe.

Come stimare in maniera ragionevole la tabella di hashing: si stima ma si ha la consapevolezza che nel tempo la struttura si adatterà in base alle operazioni e le statistiche sull'ottimizzazione del db. Le operazioni di re-indicizzazione hanno un costo importante (quando avvengono hanno dei costi importanti sul DB ma è necessario per garantire uno speed-up a seguire). Dipende anche da quanto viene utilizzato quel determinato indice

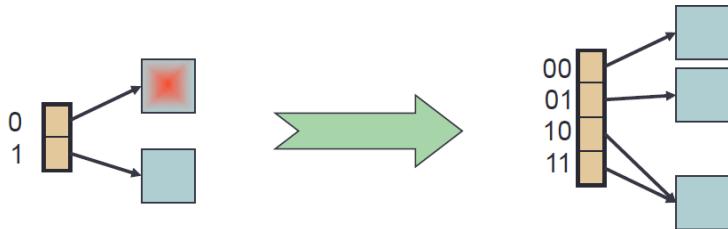
Extendible hashing: quando si ha un bucket completamente pieno (a seconda dell'implementazione vuol dire "quando il bucket è pieno oppure quando la catena di overflow è troppo lunga", ovvero "quando si reputa necessario in re-hashing") allora si raddoppia la directory, ciascun indico viene affiancato da 0 o 1 così si raddoppia MA non necessariamente si muovo tutti i dati. I dati che hanno richiesto il re-hashing vengono mediamente distribuiti, quelli che non hanno comportato questa esigenza invece rimangono invariati perché hanno ancora spazio. I punti deboli: quando un bucket è pieno (o è stata superata la soglia) si raddoppia l'intera directory, quindi è possibile che la redistribuzione dei dati nel bucket non risolve il problema perché i bucket continuano a rimapparsi nello stesso (perché rimane sempre 0 ad esempio) e quindi si continua a raddoppiare una directory (è possibile che una criticità locale comporti aggiornamenti inutili a livello globale). Nonostante i numerosi raddoppi, il problema locale persiste. Le operazioni di raddoppio e crescita esponenziale hanno un loro costo. Con il Linear Hashing si va a risolvere questo problema.

La funzione di hashing cambia quando si raddoppia la directory, raddoppiare la directory significa cambiare la funzione di hashing (nell'esempio si vede che passa da $\bmod 2$ a $\bmod 4$). La redistribuzione avviene solo

quando lo si reputa necessario e questo accade quando riempiono i buckets. Si raddoppiano gli indici e non i bucket per non raddoppiare le informazioni, i bucket non devono essere troppo piani ma nemmeno troppo vuoti (se si vuole fare una scansione sequenziale non si vuole sprecare troppo spazio).

Se i riempie il terzo bucket (il nuovo) allora si giustifica lo splitting e allora viene aggiunto un 4 bucket ridistribuendo gli indici e rimappandoli. Lo splitting avviene in maniera naturale a seconda delle **entry**.

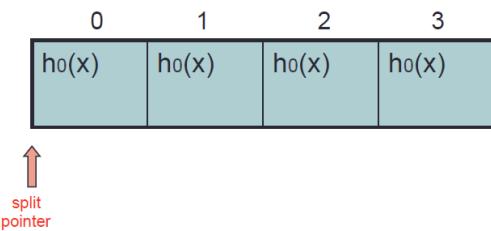
Nota: 0, 1, 00, 01, 10, 11 prendono il nome “**entry**”.



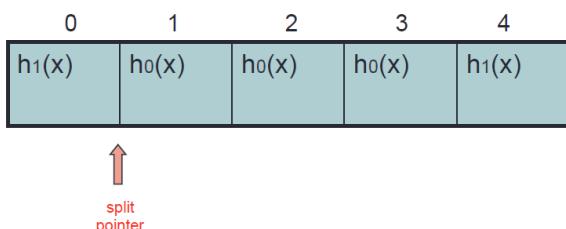
Potrebbe capitare che la directory possa diventare troppo grande perché raddoppia ogni volta che si estende (crescita esponenziale) e quindi potrebbe esserci una concentrazione locale.

Linear hashing: si ipotizza una capienza 1000 per ogni directory, se si supera la soglia del 40% si può raddoppiare tutta la directory ma invece in questo caso si raddoppia solo un bucket facendo lo split e sostituendo l'operazione con **mod 2N**. Tutti i nuovi bucket verranno associati alla stessa funzione di hash (h_1).

- Augment the hash table one slot at a time
- Two hash functions and a split pointer
 - $h_0(x) = x \bmod N$
 - $h_1(x) = x \bmod 2N$



Al seguito di uno split:



Nota:

- Se si applica h0 e si ottiene 2: ok, è stato trovato il bucket.
 - Se si applica h0 e si ottiene 0 (che è a sx dello split pointer): si applica h1 (quindi si applicano entrambe).
-

calcolo hash con h1

se hash < split pointer:

calcolo hash con h2

accedo il bucket con l'hash

Operations to Compare

Verranno confrontati i costi delle seguenti operazioni:

- Scan: fetc all records from disk
- Equality search
- Range selection
- Insert record
- Delete record

Cost Model for Our Analysis: i costi stimati ignoreranno la CPU. Costo delle diverse operazioni per accedere a un file di dati.

s, for simplicity:

- **B:** The number of data pages in the data file
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

Nota:

- record a lunghezza fissa (se fossero record a lunghezza variabile, R sarebbe la stima).
- si assume che il tempo medio di read/write sia identico in entrambi i casi.

Assumptions in Our Analysis: [riguardare questa parte della lezione]

- Heap files: la selezione è uguale per ogni chiave.
- Sorted files: i file ordinati vengono mantenuti compatti dopo le cancellazioni, quindi ordinati dopo gli inserimenti e compatti dopo le cancellazioni
- Indexes:

- Alt
- Hash: no overflow buckets (massima capienza delle pagine all'80%, ovvero per ogni pagina si ha il 20% sprecato e quindi si spreca $\frac{1}{4}$ del tempo -> la dimensione del file deve essere 1.25 data size). Formula: $1 + (\% \text{ sprecata} / \% \text{ effettivamente occupata})$
- Tree: occupazione al 67%

Assunzioni (condizionate):

- Scans:
 - Leaf levels of a tree-index are chained.
 - Index data-entries plus actual file scanned for unclustered indexes.
- Range searches:
 - We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

Costi operazioni:

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	D(log 2 B + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	D(log F 1.5B + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	D(1 + log F 0.15B)	D(log F 0.15B + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

- **Heap:** file non ordinato
 - Scan: **BD**, costo per leggere l'intero file
 - Equality: costo **0.5BD** perché:
 - caso ottimo: la chiave è nella prima pagina
 - caso pessimo: la chiave è nell'ultima pagina
 - caso medio è nel mezzo, quindi 0.5. Questo perché lo heap NON è ordinato, mediamente bisogna scandire metà chiavi
 - Range: **DB** perché avendo un file non ordinato bisogna necessariamente scansionare l'intero file
 - Insert: **2D** perché si effettua una lettura (leggere la pagina e portarla in memoria centrale) prima di inserire (scrittura) un record all'interno del file (due operazioni I/O)

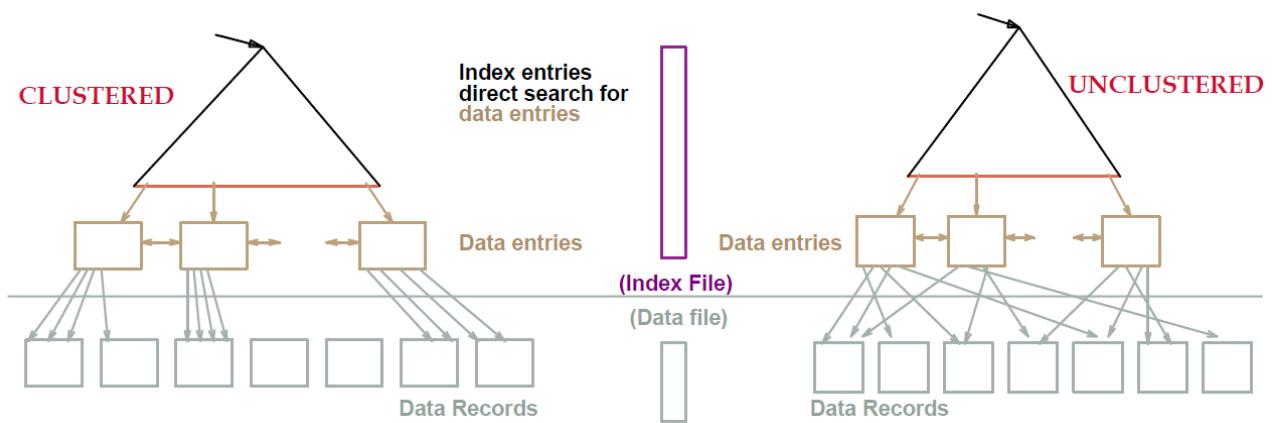
- Delete: **Search+D** perché bisogna prima cercare il record per poterlo cancellare. La cancellazione implica la ricerca che mediamente equivale al costo di **equality** per fare una ricerca per uguaglianza, quindi 0.5BD.
- **Sorted:**
 - Scan: **BD**, è possibile che il D medio di un file sorted sia più basso perché nei sorted le pagine potrebbero essere contigue (consapevolezza nei casi pratici, rimozione dei tempi di seek).
 - Equality: **Dlog2B** perché è un albero ordinato di ricerca. Sfrutta l'ordinamento del file mediante tree, quindi si effettua una ricerca binaria.
 - Range: **D(log2B + #pgs with mach recs)** sfrutta l'ordinamento del file, non bisogna più scandire l'intero file MA, dato il primo elemento si somma il costo del record dei match successivi)
 - Insert: **Search + BD**, ricerca perché è sorted, B perché bisogna mantenere compatto il file, D il costo medio di accesso. $BD = \frac{1}{2} BD$ per la lettura + $\frac{1}{2} BD$ per la scrittura. $\frac{1}{2}$ perché nel caso medio l'inserimento arriva al centro e quindi è necessario spostare metà file.
 - Delete: **uguale a insert** perché si assume che il file debba essere compatto e quindi nel caso medio bisogna spostare metà file
- **Clustered index:** struttura ad indice per cui c'era una corrispondenza (in termini di relazione di ordinamento) tra i puntatori al livello delle foglie e gli indirizzi delle pagine stesse puntate da questi puntatori, ovvero l'ordinamento dei puntatori corrisponde all'ordinamento delle pagine puntate (non ci sono puntatori che si incrociano). Trattandosi di un indice di tipo gerarchico, si assume che l'occupazione dei dati sia maggiorato del 50%
 - Scan: **1.5BD**, perché si porteranno in memoria delle pagine che sono 1/3 vuote. Perché l'occupazione è stimata al 67%, si spreca il 33 che è il 50% quindi 1.50BD.
 - Equality: **D log f 1.5 B**, il numero di accessi che si fanno corrispondono alla profondità dell'albero che è Log di base fanout del numero di foglie. f sta per fanout.
 - Range: **D(log f 1.5 B + #pgs with mach recs)**
 - Insert: **Search + D**, con sorted (tutte le pagine piene, con le aggiunte bisogna necessariamente spostare metà file) ora invece c'è 1/3 vuoto e si sfrutta questa parte nell'inserimento. Con sorted ciascun inserimento comporta lo spostamento nelle pagine perché sono tutte piene. Con Cluster invece si sfrutta l'1/3 disponibile e quindi dopo l'inserimento di avrà 1/3 - 1 libero che fa da margine.
 - Delete: **Search + D**
- **Unclustered Tree index:** il livello delle foglie dell'indice deve contenere tutte le chiavi di ricerca del file indice e per ogni chiave ci deve essere l'entry col puntatore al file di dati.
 - Scan: **BD(R+0.15)**, è un file di heap (se non interessa l'ordinamento è BD) se invece si vuole una scansione ordinata. I dati memorizzati nel file sono data entries, l'occupazione delle data entries è pari al 10% dell'occupazione delle stesse foglie se si avesse il record completo. 0.15 è il 10% di 1.5. **0.15B** è l'occupazione del file delle foglie che sto considerando. Per fare una scansione ordinata del file bisogna scandire ordinatamente tutte le foglie del file. Per ciascun record bisogna potenzialmente fare un salto, quindi bisogna leggere R volte le pagine contenenti i dati (**BR**). **(0.15B + BR) D = BD(R+0.15)**. R è il costo extra del puntatore. Questo se si vuole fare una scansione ordinata del file.
 - Equality: **D(1 + log f 1.5 B)**
 - Range: **D(log f 0.15 B + #pgs with mach recs)** come nei cluster MA cambia la dimensione che passa al 10%.
 - Insert: **Search + 2D**, 2D perché bisogna ristrutturare sia l'indice che la pagina. Il Search mette in evidenza il salto in più per accedere al dato. Accesso alla foglia, dalla foglia al dato e poi modifica (a fronte dell'inserimento, aggiungere nelle foglia e nella pagina del dato). Il Search

porta a individuare la foglia (accesso al dato), l'inserimento è da fare sia nella foglia che nel file (due aggiornamenti da fare, quindi 2D).

- Delete: **Search + 2D**. aggiorna sia la pagina del file che l'indice per dire che non c'è più

- **Unclustered hash index:**

- Scan: **BD(R+0.125)**,
- Equality: **2D**, è il punto di forza dell'hash
- Range: **BD**, non supporta l'ordinamento quindi bisogna necessariamente scansionare l'intero file
- Insert: **Search + 2D**, come il precedente MA cambia solo i costi di search (che equivale a equality)
- Delete: **Search + 2D**



Buffer Management

BMS: Buffer Management Systems.

Quando un'applicazione necessita di dati che sono contenuti all'interno di una relazione, gli scambi in lettura/scrittura (gli accesi ai dati stessi) non avvengono mai direttamente su disco ma portando il dato preventivamente in memoria centrale in quello che è un frame del buffer (è un'area della memoria centrale in cui avvengono tutte le operazioni). Quando un DBMS richiede di accedere a una pagina, la richiesta della pagina di interesse viene inviata non direttamente al disco ma al **buffer manager** (il modulo che gestisce il buffer), se la pagina è già presente in memoria centrale la rende disponibile mentre se non è già presente la porta in memoria centrale e la inserisce in uno slot libero se c'è oppure in uno slot che va a liberare. È al quanto normale dover liberare uno slot per renderlo disponibile a un'altra pagina perché la memoria centrale è nettamente più piccola rispetto a quella secondaria.

È necessario di ottimizzare il numero di accessi al disco che l'uso delle strutture dati che si rendono necessarie per la sua volta ottimizzare l'uso del disco. Compromesso tra il costo di gestione per avere tutte le informazioni necessarie (avere la strategia ottima) e la strategia ottima di per sé.

trade-off per avere una strategia ottima e la strategia ottima di per sé.

Politiche di rimpiazzamento del buffer: scelta della pagina vittima qualora arrivi al buffer manager la richiesta di una pagina che non è in RAM e al momento non può essere portata in RAM a meno di liberare spazio poiché satura.

- **Scelta a caso:** politica meno costosa. Il punto forte è che è molto veloce, non ha bisogno di strutture ausiliarie o algoritmi sofisticati. Il punto debole è che è poco efficiente, ad esempio potrebbe essere riletta tantissime volte una pagina che viene in continuazione rimossa poiché viene spesso scelta per rimuoverla a fronte di un'altra aggiunta (le letture disco dal buffer potrebbero essere maggiori perché non c'è una strategia per scegliere le pagine che magari nell'immediato futuro non vengano utilizzate, è una questione di probabilità). Non c'è una **stima (previsione)** dell'utilizzo futuro delle pagine sapendo come sono stati implementati gli algoritmi. La scelta casuale non è per niente guidata dalla conoscenza di quello che succederà a seguire, potenzialmente si può decidere di rimuovere una pagina che serve nell'immediato futuro (fosse stato scelto un altro candidato questo non succederebbe).

Nota:

- Se la pagina rimossa è stata modificata in memoria centrale, il costo della rimozione comprende anche la scrittura su disco.
- Le scelte di rimpiazzamento non avranno solo un impatto locale ma anche sul controllo degli accessi e gestione della concorrenza, interella quindi anche altri moduli che non appaiono come immediatamente collegati.
- Avranno anche impatto sulle politiche di controllo degli accessi (**concurrency control**), gestione della concorrenza, perché è possibile avere certe pagine condivise da più query. Per la gestione della concorrenza si ricorre spesso ai **lock** che in scrittura sono esclusivi mentre in lettura no, è possibile avere quindi diverse transazioni che stanno accedendo a dati che stanno nella stessa pagina e quindi l'azione che si svolge su quella pagina può avere impatti positivi/negativi su più di una transazione che portano le conseguenze della scelta di usare quella pagina come candidata per il rimpiazzamento.
- Anche il **crash recovery** comporterà comportamenti diversi del modulo per la gestione del crash, in particolare se le pagine sono in memoria centrale o sono già state salvate su disco.
- Il buffer manager è quindi un modulo “cuore” per il sistema e da lui dipendono i comportamenti di tanti altri moduli anche se, apparentemente, possono sembrare non collegati.

Nozioni importanti per una gestione adeguata del buffer:

- **Pin counter:** un contatore che conta il numero di transazioni che in un certo istante stanno utilizzando una specifica pagina contenuta in uno specifico frame del buffer. Se il buffer avrà 500 frame, avrà 500 pin (ognuno corrispondente a una pagina del buffer). È gestito dal **transaction manager** (insieme al buffer manager) per mantenerlo sempre aggiornato e corretto.
- **Dirty bit:** per ogni frame del buffer c'è un bit che dice se la pagina è “sporca”, ovvero diversa per effetto di modifiche rispetto alla pagina su disco. Il fatto che la pagina sia stata modificata da indicazione sull'opportunità di salvarla su disco a fronte di rimpiazzamento della stessa. Se non è stata modificata non ha senso riscriverla su disco mentre se è stata modificata bisogna renderla persistente su disco per gli utilizzi futuri.

Quindi le pagine se si scelgono per il rimpiazzamento, se hanno il **dirty bit** pari a uno devono essere salvate su disco prima che il corrispondente spazio venga reso disponibile a un'altra pagina.

Supponiamo che la transazione che sta operando sulla pagina che è stata identificata come vittima (su cui sono già state fatte delle modifiche) subisca un crash, in questo caso la pagina che è stata modificata deve o **non** deve essere salvata su disco a seconda se la transazione, a cui fa parte, avesse fatto **commit** o meno. Fare commit vuol dire aver dichiarato di aver concluso i dati nuovi e per via della condizione di durabilità (**durability**), che deve valere sui database relazionali, le operazioni che hanno fatto commit devono diventare persistenti. Se un dato è su una pagina e la corrispondente transazione ha fatto commit, è giusto che questa vada poi su disco. Se si salvano su disco dei dati provvisori di una transazione che non è stata ancora completata e che farà crash (quindi dovrà essere disfatta e non conservata e resa durevole), l'effetto del salvataggio su disco di questa pagina parzialmente modificata è un'operazione di **rollback** ovvero il disfacimento di quello che era stato reso ufficiale prematuramente.

Il **rollback** può essere quindi un'operazione costosa con degli effetti collaterali non trascurabili, si ribadisce quindi la necessità di operare con strategie il più possibile intelligenti e guidate dalla conoscenza che si ha per ridurre al minimo la casualità e la scelta di vittime che potenzialmente comportino scritture che nell'arco di breve tempo devono essere rifatte.

Transaction manager: gestire il pin counter, insieme al buffer manager, mantenendo l'informazione costantemente aggiornata e corretta.

Responsabilità del buffer manager:

- Mantenere una tabella di associazione degli identificatori delle pagine sul disco e i frame sul buffer, ovvero mantenere correttamente il mapping tra l'indirizzo fisico delle pagine su disco (che è l'identificatore della pagina sul disco) e il frame del buffer in cui è eventualmente memorizzata. Quando l'utente fa una query, la sua query fa riferimento a dati memorizzati in una certa pagina di disco. L'indice ha i puntatori rispetto all'indirizzo fisico di memorizzazione della relazione su disco, allora il buffer manager deve saper dire se la pagina N c'è o non c'è, se c'è in quale frame si trova.
- Deve avere l'informazione su quali frame sono disponibili per nuove letture. Si cerca la pagina N, non c'è nel buffer e quindi si identifica un frame libero e poi si aggiorna correttamente la tabella del mapping che dice chi si trova dove. In assenza di frame liberi deve avere una sua strategia di rimpiazzamento per scegliere chi liberare in caso di necessità.

Note:

- In assenza di informazioni, si usa tipicamente una politica del tipo LRU, o in alternativa MRU.
- **LRU** e **MRU** rispondono a un principio di località temporale, ossia assumere che in un certo intervallo di tempo ci sia una tipologia di pagine che vengono utilizzate di più.

Most popular replacement schemes (sono delle stime/ipotesi):

- **LRU:** Least Recently Used (o in alternativa Least Frequently Used)
 - **LRU:** sceglie la pagina usata meno di recente
 - **LFU:** sceglie la pagina che viene usata più di rado

e che probabilmente ora non ha più ragione di esistere (probabilmente non serve più) nella speranza che non servi nell'immediato futuro.

LRU e LFU sono politiche molto utilizzate anche nei SO che corrispondono a un principio di **località temporale**. Questo vuol dire assumere che all'interno di un certo intervallo di tempo ci sia un certo gruppo di pagine più utilizzate per cui se una pagina è stata portata in memoria viene utilizzata per un bel po' (magari tante operazioni) poi si completano le operazioni che fanno riferimento a questa pagina, a lavoro finito questa pagina per un bel po' non servirà fino a quando non verrà riproposta un'altra query che la riguarderà. Per cui è un principio di **località temporale** perché una pagina serve con una certa frequenza all'interno di un certo intervallo di tempo per poi diventare ridondante e inutile nel buffer perché non utilizzata dalle transazioni a seguire.

Nota: se si ha ragione di ipotizzare che valga la **località temporale**, allora la politica LRU è adeguata. Si può presentare il problema del **sequential flooding** (ogni aggiunta comporta un page fault). LRU è molto utilizzato ma non adeguato a tutti i pattern di accesso.

- **MRU:** Most Recently Used, approccio duale. Corrisponde a un principio di **anti-località temporale**, ovvero se si ha appena usato una pagina allora per un po' non servirà più e quindi si privilegiano le più vecchie che potrebbero tornare utili. È utile quando si presenza un sequential looping (accesso sequenziale alle pagine) come pattern di accesso, a differenza di LRU che comporterebbe il sequential flooding.
- **Clock:** A heuristic that approximates LRU. Pone rimedio ai costi dei singoli passi precisi perché, ad esempio, per scegliere la pagina LRU bisogna avere un ordinamento degli accessi a tutte le pagine e quindi bisogna gestire una valida struttura dati (tipicamente una coda con priorità) che consenta di restituire l'informazione interessata tenendo ordinate tutte le pagine e frame del buffer rispetto all'istante di utilizzo. Siccome le pagine sul buffer sono tante e le attività sul buffer è frenetica perché tutte le operazioni avvengono lì, la gestione di tutte le operazioni e il mantenimento aggiornato della coda di priorità è molto costoso. Per questo motivo, spesso quando si sceglie di adottare questa strategia non la si adotta fedelmente ma la si emula attraverso altri algoritmi come quello del clock. L'algoritmo del clock, a fronte di costi decisamente più contenuti emula LRU e quindi cerca di restituire e riconoscere pagine che non sono utilizzate da tanto tempo (non necessariamente quella utilizzata da più tempo perché il costo della gestione della struttura rallenterebbe in maniera importante l'individuazione del candidato e quindi il beneficio dato dalla politica adeguata lo si pagherebbe in termini di gestione della politica stessa).

Se nel caso specifico emula bene LRU, non risolve il sequential flooding ed eredita tutti i pregi e difetti di LRU.

- LRU is expensive

Nota: la scelta migliore è data dalla situazione. LRU è molto costoso perché bisogna mantenere gli accessi delle pagine, per questo quando lo si applica generalmente non lo si replica fedelmente ma lo si emula attraverso il clock (euristica di LRU). Trade-off tra vantaggi e costi della politica stessa.

Sequential flooding: inondazione sequenziale dovuta al fatto che si “cicla” per tante volte sulla sequenza di pagine perché ogni volta si sostituisce quella che serve. Ogni volta che si riempie la memoria, ogni nuova lettura/aggiunta comporta sovrascrivere un’altra pagina.

In assenza di informazioni sul pattern di accesso, LRU può essere una strategia conveniente. Se si ha l’informazione che le pagine vengono usate in maniera ciclica (pattern di accesso), allora MRU è quella che risulta più conveniente. Ipotizzando un buffer che può contenere 3 pagine e 4 pagine richieste, con LRU verrebbe rimossa sempre la pagina che serve nell’operazione successiva mentre con MRU si va incontro a questo problema.

Frame vs pagine: il buffer è un array di frame ciascuno dei quali ha una dimensione pari alla dimensione di una pagina del disco, per cui ogni frame contiene una pagina di disco.

Clock algorithm: si chiama così perché itera in maniera ordinata e ciclica dal primo all’ultimo frame come se fosse un orologio. Per scegliere la pagina vittima c’è un puntatore che opera come la lancetta di un orologio che via a via si va a posizionare su tutti i frame del buffer. Si vuole scegliere come vittima quella che non è stata usata da molto tempo.

Informazioni che servono:

- **pincount:** pincount > 0. dice quante sono le transazioni attive sul buffer, se è > 0 vuol dire che quel frame del buffer è in uso.
- **referenced:** pincount = 0; referenced = 1. Il bit referenced è il bit che si utilizza nel visitare le diverse pagine. A seconda delle versioni dell’algoritmo, il **bit** è inizializzato a 0 o 1.
- **available:** pincount = 0; referenced = 0.

L’idea è quella di scegliere pagine che non sono state utilizzate almeno per la durata di un ciclo completo su tutte le pagine del buffer. La pagina è inizializzata con il reference =1 (rispetto al lucido), quando la pagina è “non pin” e quindi ha il pincount =0 (che vuol dire che in questo momento non la sta utilizzando nessuno) ma ha il referenced =1 (**stato: pincount = 0; referenced = 1**) vuol dire che è candidabile ma non disponibile, quindi la si dichiara available ponendo referenced =0 (**stato: pincount = 0; referenced = 0**). Al prossimo giro dopo averle visitate tutte, se la pagina non è stata selezionata da nessuno e continua a rimanere nello stato “available”, allora viene scelta come vittima perché è potenzialmente rimasta non utilizzata per un intero giro. L’idea è di scegliere le pagine che sono state vuote almeno il tempo di fare tutto il giro completo delle pagine nel buffer. La pagina scelta potrebbe non esser stata utilizzata oppure utilizzata in maniera talmente veloce che è di nuovo disponibile.

In poche parole, l’algoritmo del clock cerca di approssimare un comportamento che gestisce la frequenza dell’utilizzo delle pagine senza confermare effettivamente l’informazione della frequenza d’uso delle stesse.

- pinned: pincount >0
 - referenced: pincount=0; referenced=1
 - available: pincount=0, referenced =0
1. loop until (found eligible page) or (all pages pinned)
 1. If page[current] = pinned
 1. current++
 2. If page[current] = referenced
 1. reference = 0; current++
 3. If page [current] not referenced and not pinned
 1. replace page
 2. referenced=1;
 3. current++;

Alg (to check):

```

WHILE true
(1) Obtain the candidate buffer descriptor pointed by the nextVictimBuffer
(2) IF the candidate descriptor is unpinned THEN
(3)   IF the candidate descriptor's usage_count == 0 THEN
        BREAK WHILE LOOP /* the corresponding slot of this descriptor is victim slot. */
    ELSE
        Decrease the candidate descriptor's usage_count by 1
    END IF
END IF
(4) Advance nextVictimBuffer to the next one
END WHILE
(5) RETURN buffer_id of the victim

```

Qualche stima sul tipo di accessi per il futuro si possono fare e quindi ci si chiede come si possono migliorare le prestazioni del buffer manager e come si possono sfruttare queste conoscenze che si hanno in modo da avere impatto sull'efficienza e l'efficacia della strategia di rimpiazzamento.

Criteri/approcci che si possono utilizzare:

- **Separazione dei domini:** ossia anziché trattare tutti frame di buffer in modo equivalente, si trattano porzioni di buffer diverse con diverse tipologie di file e si adottano diverse strategie di rimpiazzamento/uso delle pagine che costituiscono i file adattandole allo specifico tipo di file che è contenuto nella specifica porzione del buffer. L'idea di separare in base ai domini ha il vantaggio di tenere conto della specificità del tipo di dati e del modo con cui questi vengono utilizzati data la natura dell'organizzazione gerarchica dei dati stessi. Questa scelta è già una buona approssimazione. È una strategia abbastanza utilizzata.

Dato che, ogni volta che si accede ad un albero, il primo nodo che viene navigato è la root, questo è il nodo che viene utilizzato di più. Si può pensare di separare la porzione di buffer dedicata agli indici dalla porzione destinata agli altri tipi di file (heap, sorted, ...) e, per quanto concerne gli indici, privilegiare (che vuol dire scalzare con minor probabilità) quelle pagine che si presume saranno utilizzate molto di più. In questa situazione, è ad esempio visibile come vengano trattate con una

politica diversa le pagine destinate alle radici degli indici (o comunque a quelli di livello alto), rispetto a quelle che sono le pagine destinate alle foglie.

L'ultimo accesso fatto all'indice è tipicamente una foglia, quindi se bisogna andare a ricoprire il nodo di un indice, quello usato più lontano nel tempo è tipicamente la radice perché la scansione è avvenuta root -> leaf **ma** allo stesso tempo la root è anche quella che si sa che servirà al prossimo utilizzo dell'indice stesso. In questo caso il LRU avrebbe di nuovo un effetto contro-producente, mentre potrebbe essere vantaggioso sulla rimanente parte del buffer perché magari, per quanto concerne i dati delle relazioni vere e proprie (non gli indici), un accesso guidato da LRU potrebbe essere quello vantaggioso.

- **Algoritmo del working_set:** prevede che la priorità di una pagina dovrebbe essere determinata sulla base della frequenza rispetto a cui viene utilizzata la relazione di cui la pagina fa parte. Ad esempio, se si ha una relazione memorizzata in 5 pagine diverse (alcune delle quali sono già eventualmente nel buffer), a ciascuna delle pagine verrà associata una priorità che verrà tenuta in considerazione dalla politica di rimpiazzamento in particolare candiderà a essere rimpiazzate le pagine che hanno priorità più bassa. La priorità di questa pagina è data dalla frequenza con cui la relazione, che contiene quella pagina, viene utilizzata (ad esempio se nel buffer c'è un certo numero di pagine tutte provenienti dalla stessa relazione, queste avranno tutte la stessa priorità che è dettata dalla frequenza di accesso alla relazione di cui queste pagine fanno parte). L'idea è quella di non utilizzare genericamente LRU e MRU, si associa alle pagine che non sono di indice una priorità che è la frequenza d'uso di una relazione di cui queste fanno parte.

Il problema di LRU in merito al costo della gestione nella frequenza, in questo caso è molto ridimensionato perché nel primo caso si presenta la necessità di gestire una coda con priorità che ha tanti elementi quanti sono i frame del buffer e a ogni singolo accesso si doveva eventualmente riordinare tutta la coda (questo è drammatico). Il numero di relazioni presenti nello schema del database è molto più contenuto, quindi la gestione di una coda che tenga ordinata la priorità sulla base della frequenza di accesso alle relazioni stesse costa decisamente meno ed è decisamente più facilmente contenibile.

Sulla base di queste priorità che viene associata alle diverse relazioni, si associa a ciascuna relazione (in base alla sua priorità) un certo numero di frame di buffer. Quando arrivano le query (quando servono le pagine), se c'è ancora disponibilità di pagine all'interno del buffer (può succedere quando tutte quelle utilizzate precedentemente sono state rilasciate perché non più attive) allora si acquisisce la pagina senza alcun rimpiazzamento. Quando non ci sono frame disponibili, ciascuna transazione che opera su una certa relazione rimpiazza applicando la sua politica di rimpiazzamento rispetto alla porzione di buffer che gli è stata assegnata.

Quindi si ha l'ordinamento delle relazioni, la priorità associata alle relazioni che determina il numero di pagine di buffer associata a ciascuna relazione, la gestione del buffer locale alle singole relazioni e a livello locale l'utilizzo della politica MRU che è quella che garantisce di limitare i casi di sequential flooding che invece si andrebbero potenzialmente a verificare con LRU.

Problemi: si usa MRU perché è quello che evita il sequential flooding ma **non** è sempre la politica ottimale perché si è visto che in alcuni casi risulta più opportuno utilizzare LRU quando vale il principio di località temporale. Il fatto di utilizzare per ciascuna relazione sempre la stessa politica

può essere vantaggioso o meno, per questa ragione in alcuni casi la tecnica basata su working set (algoritmo new) non sempre funziona bene.

L'algoritmo del *working_set* prevede, a livello di singoli *working_set*, sempre l'applicazione della stessa politica. C'è una separazione dei domini, una sorta di gestione localizzata dello spazio associato alle singole relazioni ma non c'è una "personalizzazione" rispetto al tipo di operatore che si sta implementando (per alcuni operatori LRU è la politica ideale mentre per altri MRU). Questa è una limitazione dell'algoritmo.

- **HotSet:** si chiama così perché è basato sull'idea di hot point (punto caldo) che è il più piccolo numero di frame che si può allocare alla query garantendo un drastico calo del page fault che ottengo rispetto ai precedenti, ovvero si calcolano i page fault con 1 frame, 2, 3 ecc. fino a che si trova un numero "piccolo" che consente un calo drastico dei page fault (ci sono volte in cui l'aggiunta di un solo frame fa la differenza).

Naturalmente, operatori diversi (ho implementazioni diverse dello stesso operatore) possono avere hotset diversi. Se bisogna fare una scansione sequenziale del file, ad esempio per fare una selezione su un campo non indicizzato, si leggono i record della prima pagina per verificare quali vengono richiesti dalla query e poi si sovrascrive/sostituisce la pagina corrente con la successiva e così via (per questo tipo di implementazione l'hotset è 1 perché basta un solo frame di buffer per ridurre al minimo i page fault, rimangono solo quelli strettamente necessari per portare in memoria le pagine una volta sola). Ad esempio, nel caso del join i record vengono più volte quindi avere più frame potrebbe ridurre i page fault.

L'algoritmo dell'HotSet ricorda per certi versi quello basato su *working_set* ma in questo caso il buffer non è allocato ai singoli file MA la porzione del buffer è assegnata alle singole query. La quantità di buffer e il modo in cui viene gestito il buffer associato alle singole query è determinato (a grandi linee) dal comportamento della query stessa per quanto concerne i cicli, quindi dal fatto che l'implementazione dell'operatore, su cui si basa la query, richieda che si cicli o meno. Per i diversi operatori esistono diverse implementazioni differenti. In base al tipo di implementazione e all'operatore, è opportuno assegnare una certa quantità di buffer alla query stessa oppure ne basta meno.

Nota: qui si nota un'altra stretta connessione tra i diversi moduli perché se si userà un certo tipo di operatore (ad esempio il join) servirà un certo numero di frame del buffer per evitare un eccessivo numero di page fault. Ad esempio: se all'operatore join vengono riservati 4 frame di buffer, l'ottimizzatore sceglie quale implementazione dell'operatore deve adottare tenendo conto delle risorse su cui può contare. Nel momento in cui si conoscono l'operatore che è alla base della query e l'implementazione dell'operatore, è relativamente facile stimare, almeno in forma pessimistica, quello che sarà il numero di letture da disco necessario e di conseguenza il numero di page fault che si renderanno necessari data una previsione dell'allocazione di specifici frame di buffer.

Problemi/debolezze:

La stima avviene sulla base del caso peggiore senza conoscere in termini molto precisi (perché sarebbero troppo costose mantenerle) le statistiche legate alla distribuzione dei valori degli attributi. Per essere troppo cauti spesso si va ad allocare un numero eccessivo e quindi si presenta uno spreco

- **Query Locality Set Model:** si basa sull'insieme di relazioni locali alla query che si sta considerando, è un approccio che cerca di "far cadere il peggio" dei precedenti. Tiene conto sia dell'aspetto legato all'hotset che da quello legato alle relazioni su cui la query sta operando.

Identifica i piani di valutazione della query (**query plan**), una formulazione procedurale del piano di valutazione della query stessa e quindi della sequenza di operatori da applicarsi con un'indicazione specifica della loro implementazione scelta. All'interno di un query plan, identifica un certo numero limitato di pattern di accesso diversi (la prevedibilità è il grosso punto di forza del modello relazionale perché si ha un numero limitato di operatori, di implementazioni per operatore e di pattern di accesso che consentono di implementare tutti gli operatori). I pattern di accesso sono le strategie di visita delle pagine delle relazioni coinvolte dall'operatore, indicano l'ordina di lettura delle pagine su cui si sta andando ad operare.

Pattern di accesso:

- **straight sequential (ss):** accesso “banalmente sequenziale”, apertura del file e scansione sequenziale pagina per pagina una volta sola. Legge una pagina poi la sostituisce con la seconda ecc., è sufficiente solo un frame e nessuna pagina verrà letta più volte.

Requisiti di utilizzo di frame del buffer: 1 frame.

- **clustered sequential (cs):** è sequenziale, ricorda il precedente, MA prevede che si apra un file, si utilizzi ciascuna pagina una volta sola e sequenzialmente nella scansione del file e prevede anche che, per lo specifico operatore che si sta considerando, ad un certo punto debbano essere presenti in memoria più di una pagina (un cluster di pagine, ovvero un gruppo di pagine) ai fini dell'operatore che si sta implementando. Ad esempio, il **sorted merge join** prevede che siano contemporaneamente presenti dei blocchi consecutivi di pagine del file (in questo caso un solo frame non sarà sufficiente).

Requisiti di utilizzo di frame del buffer: la dimensione del cluster determina il numero di frame del buffer.

- **looping sequential (ls):** è il pattern che si utilizza nei meccanismi di join, è il pattern che si utilizza nella relazione interna. È come lo SS nel senso che si legge una pagina per volta in maniera sequenziale MA al termine del file si inizia di nuovo da capo e si cicla sul sequenziale. Si può utilizzare una sola pagina ma al costo di dover rileggere la stessa pagina una volta per iterazione del ciclo. LS è il tipo di pattern d'accesso che si ha sulla relazione interna del join nell'implementazione naïve del join.

Requisiti di utilizzo di frame del buffer: se il file fosse sufficientemente piccolo da essere tutto contenuto in memoria centrale, il numero ideale di frame sarebbe pari alla dimensione del file (ovviamente è una politica greedy, pertanto da mantenere sotto controllo). Più frame e meglio è però è da tenere sotto controllo. Più pagine si riescono a dare alla query (ovvero un numero \leq alla dimensione della relazione, oltre sarebbe uno spreco) che deve eseguire un LS pattern, meno saranno i page fault a cui si andrà incontro. Quello che è importante è che si danno più frame possibili e poi questa collezione di più possibili pagine che si assegnano alla query si gestiscono con la politica di rimpiazzamento MRU (LRU provocherebbe sequential flooding, siccome si sta ciclando in maniera sequenziale l'ultima pagina letta è quella che servirà più lontano nel tempo).

- **independent random (ir):** aprire il file e, in un modo casuale, andare ad accedere ad una pagina mediante un meccanismo di accesso diretto e la pagina a cui si accede è indipendente dalle precedenti. Non c'è più ordinamento sequenziale (le pagine sono in ordine sparso, ad

esempio viene chiesta la pagina 1, la 7, la 12, la 3, ecc.). Può capitare ad esempio quando l'accesso ai dati contenuti nel file avviene mediante un indice non clustered, la nuova pagina è potenzialmente indipendente dalla precedente. Prevede che il punto di inizio dei dati a cui si è interessati sia individuato mediante accesso diretto, quindi in modo casuale e indipendentemente dal precedente. LRU.

Requisiti di utilizzo di frame del buffer: 1 frame.

- **clustered random (cr):** simile al precedente perché è random con la differenza che può capitare che nel momento in cui con accesso diretto si arriva ad una certa pagina, è possibile che serva portare in memoria un cluster di pagine (un gruppo) per averli contemporaneamente presenti. È la versione cluster dell'operatore IR che ha bisogno dell'uso di più record simultaneamente. Prevede che il punto di inizio dei dati a cui si è interessati sia individuato mediante accesso diretto, quindi in modo casuale e indipendentemente da quello che è stato fatto nel cluster precedente.

Requisiti di utilizzo di frame del buffer: il numero di frame che servono è pari alla dimensione del cluster, ovvero al numero di pagina che bisogna poter conservare simultaneamente in memoria.

Fino a qua ci sono i pattern di accesso che si possono avere sui normali file di dati ma oltre a questi si opera “pesantemente” anche sui file di indici, di seguito i pattern di accesso:

- **straight hierarchical (sh):** “banalmente gerarchico” (gerarchico in senso stretto), parte dalla radice e sequenzialmente segue la gerarchia per arrivare alla foglia a cui si è interessanti quindi si leggere il node/la pagina che corrisponde alla radice e, analizzando il contenuto della radice, si identifica la pagina successiva che corrisponde al puntatore relativo al range della chiave che si sta cercando, sequenzialmente si leggerà il nodo di livello due fino ad arrivare alla foglia. Percorre sequenzialmente il cammino dell'albero che porta dalla radice alla foglia interessata.

Requisiti di utilizzo di frame del buffer: 1 frame (operando a livello di query locality set, ovvero a singola query e non a più query simultanee sullo stesso indice).

- **sh+ss:** combinazione dei due approcci SH e SS, SH lo si adotta ad esempio per le query di range. B+ tree is sorted and sequential. Seguendo lo SH si arriva al nodo che contiene l'inizio dell'intervallo interessato e poi, essendo l'indice ordinato, sequenzialmente attraversa i “fratelli” del nodo per andare ad includere/riconoscere tutto il range interessato. L'attraversamento dell'indice è gerarchicamente sequenziale per raggiungere l'inizio del range e poi sequenziale “piano” (senza seguire una gerarchia) rispetto all'ordinamento delle chiavi per andare ad individuare gli elementi che servono.

Requisiti di utilizzo di frame del buffer: 1 frame.

- **sh+cs:** combinazione dei due approcci SH e CS, simile all'idea precedente. Prevede che gerarchicamente si arrivi all'inizio del range interessato e poi si vada a recuperare un insieme

di pagine da tenere simultaneamente presenti in memoria principale perché le operazioni che si vanno ad effettuare richiedono che siano presenti più di una pagina simultaneamente.

Requisiti di utilizzo di frame del buffer: il numero di frame dipende alla dimensione del cluster, ovvero dal numero di pagine.

- **looping hierarchical (lh):** idea di dover “ciclare” più volte sullo stesso indice, quindi si cicla sull’attraversamento gerarchico (root->leaf, poi di nuovo root ecc.).

Requisiti di utilizzo di frame del buffer: $\text{prob}(p \text{ in ith level}) = 1/f^i$. Prevede un comportamento che ricorda il domain separation. Se bisogna ciclare più volte sullo stesso indice, si può dire con certezza che a ogni iterazione del ciclo certamente si passa dalla radice e con probabilità “1 su fanout” si passerà a uno degli altri figli ecc. (ogni reiterazione passerà dalla radice quindi se è possibile mantenerla ben venga). Il fatto di poter riutilizzare, e quindi di non dover fare page fault, è funzione del fanout (ovvero del numero di discendenti che la pagina ha) e a volte può essere prevedibile se ci sono informazioni sulla distribuzione delle chiavi nel file e sulla probabilità che certi dati vengano cercati di più rispetto ad altri e quindi la frequenza con cui certe query vengono fatte. In questo caso le statistiche sulla distribuzione delle chiavi e sulle caratteristiche fisiche dell’indice possono venire in aiuto. Si ha la certezza che la radice venga riutilizzata (quindi conviene mantenerla) e poi via via le probabilità di riuso decrescono man mano che si discende l’indice perché aumentano il numero delle “foglie” ai diversi livelli. In questo caso la tecnica di gestione ricorda il domain separation che è alla base degli approcci descritti precedentemente.

DBMin: è un algoritmo che cerca di ottimizzare l’uso del buffer considerando i diversi pattern di accesso ai dati decidendo, per ciascun pattern di accesso, un opportuno algoritmo di gestione del buffer. Si definiscono i locality set che abbinano l’informazione relativa alla query che si sta calcolando (e quindi al suo pattern di accesso) e al file su cui si sta operando. Si va verso un’operazione che è un ibrido tra il working_set (che usa solo il file, ovvero la relazione) e l’hotset (che usa soltanto la query).

Questo metodo combina i due metodi: su questa relazione con quale modalità la query, che si sta utilizzando, sta accedendo e ciascuna associazione di una relazione alla query che la sta utilizzando viene gestita indipendentemente in base alle caratteristiche del pattern di accesso che si sta utilizzando.

Quindi a ciascun locality set si associa un certo insieme di pagine di buffer e si procede in questo modo: si lavora a livello di locality set, ovvero a livello di associazioni tra relazioni e query che operano sulle relazioni secondo uno specifico pattern di accesso gestendo ciascun locality set in modo individuale/indipendente dagli altri. Ciascun frame del buffer, ciascuna pagina presente in memoria centrale, appartiene ad un locality set (è una pagina che appartiene ad una relazione ed è in questo momento in memoria centrale per essere oggetto di valutazione da parte della query che è citata in quel locality set).

Quando la query richiede un dato che sta all’interno di una pagina di una relazione:

- Se la pagina di quella relazione è già presente in memoria (e appartiene al locality set della query stessa) la si legge e basta,
- Se la pagina è già presente in memoria ma nel locality set di qualcun’altra query che usa sempre quella pagina allora la pagina rimane lì dov’è ma virtualmente andrà a far parte del locality set della query corrente, farà parte del locality set della query che l’ha appena cercata e quindi sarà soggetta alle sue politiche di rimpiazzamento (grande condivisione, cambierà solo la politica di

rimpiazzamento). Il fatto di far parte del locality set di una query o dell'altra cambia perché ad ogni locality set, che corrisponde ad uno specifico pattern di accesso, corrisponde una politica di rimpiazzamento. Il fatto di poter cambiare locality set ha effetto sulla possibilità di applicare un cambiamento nella politica di rimpiazzamento a cui sarà soggetta (la pagina). Cambia l'owner e l'implicazione di questo cambiamento consiste nell'utilizzo di quale politica di rimpiazzamento verrà utilizzata sulla pagina.

- Se la pagina era già in memoria e non aveva alcun proprietario può essere successo che qualcuno l'abbia portata in memoria, ha finito di operare e non serve più perché non c'è una query attiva su quella pagina (quando un qualcosa viene letto non è che viene rimosso/sovrascritto). Avere un owner vuol dire che c'è una query che la sta usando o l'ha usata, non avere un owner vuol dire essere stata usata ma in questo momento chi l'ha usata non è più attivo e in questo momento non c'è più perché ha completato il suo compito. In questo caso la pagina diventa proprietà del locality set della query che l'ha richiesta.

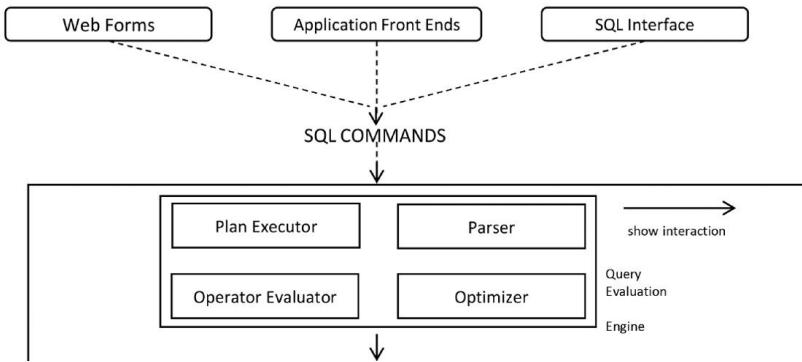
Index based query: sono quelle query che si possono valutare completamente senza necessariamente dover accedere al file di dati. Ad esempio, se si ha un file di indice sui cognomi e nomi e questo indice è non clustered, a livello di foglie si ha tutte le coppie cognome-nome ciascuna associata alla data entry relativa all'individuo. La risposta alla query avviene **solo** utilizzando l'indice, è il tipo di query che si può eseguire senza fare accesso al file di dati vero e proprio.

Lo si può fare SOLO nei file di tipo NON clustered perché negli indici non clustered a livello di foglia si hanno tutte le chiavi indicizzate mentre negli indici clustered si hanno solo degli intervalli che rimandano alla porzione di file ordinata (memorizzata in maniera contigua) che quindi contiene più informazioni di quelle presenti nel solo indice.

Esempi:

- "Restituisci il voto medio degli studenti il cui cognome comincia con E", allora bisogna posizionarsi sul primo studente il cui cognome comincia con "E" e poi andare ad accedere a tutti i record dei corrispondenti studenti (quindi accedere a tutto il file) e restituire il voto medio. Questa **NON** è una **index only query** perché sarebbe una query che richiede l'accesso ai dati.
- "Restituisci l'elenco dei cognome degli studenti il cui cognome comunque con E", questa informazione la si può restituire senza accedere ai dati perché l'elenco c'è già a livello delle foglie dell'indice. Gerarchicamente si scende alle foglie dell'indice fino a trovare il primo studente che inizia con "E" e poi sequenzialmente sulle foglie dell'indice legge e va avanti fino a trovare un cognome che non comincia più per "E".

Query evaluation



Uno dei punti di forza del modello relazione è che la query viene espressa in modo dichiarativo e tradotta in una forma procedurale che l'ottimizzatore conferma associando uno specifico algoritmo di implementazione a ciascun operatore dell'algebra relazionale (che è l'algebra procedurale su cui ci si basa). Ad esempio, il join ha tante alternative di implementazioni diverse (ciascuna con pro e contro a seconda della situazione). L'**ottimizzatore**, sulla base del caso specifico, individua il pattern.

Operatori relazionali:

- **Selezione**: data una relazione e un criterio di selezione (ossia la condizione che i record interessati devono soddisfare), record per record valuta se la condizione è soddisfatta o no per restituirlo o meno.
- **Proiezione**: data una relazione che è definita su un certo insieme di attributi, restituisce la relazione costituita soltanto da un sotto insieme di quei contenuti (quindi uno schema contenuto in quello della relazione).
- **Join**: prese due relazioni e una condizione di join, restituisce i record delle due relazioni che soddisfano la condizione.
- **Prodotto cartesiano**: operatore che crea un match tra tutti i record del primo operando e del secondo. Corrisponde al fare un join con una condizione che è sempre vera (soddisfatta da tutti i record). È uno degli operatori insiemistici, oltre a questo ne sono presenti anche altri: differenza, unione, intersezione, eliminazione dei duplicati
- **Differenza insiemistica**
- **Unione** a condizione che abbiano lo stesso schema
- **Intersezione**
- **Aggregazione**: SUM, MIN, GROUP BY ecc. Sono operatori che implicitamente richiedono un ordinamento della relazione su cui bisogna operare in alcuni casi. A esempio, GROUP BY è un operatore che raggiunge facilmente il suo obiettivo su una relazione ordinata perché se ordino rispetto all'attributo (o alla sequenza di attributi) che si ha come condizione del raggruppamento, effettivamente i dati sono naturalmente organizzati i dati da valutare in gruppi (all'interno eventualmente del gruppo vengono effettuate altre operazioni).
- **Eliminazione dei duplicati** (non è un operatore che non si applica mai da solo, non ha dignità a se stante, ma indirettamente si richiama ad esempio quando si fa un DISTINCT in una query SQL), è un operatore ausiliario ed è tra i più costosi, si limita l'applicazione di questo operatore solo se l'utente lo chiede. Per definizione una relazione è un insieme di record e un insieme non ha duplicati, però la relazione che si ottiene quando si interroga il DB contiene duplicati (a meno della clausola DISTINCT specificata dall'utente) e quindi teoricamente non sarebbe più una relazione perché eliminarli sarebbe costoso.

- **Sort:** operatore di ordinamento, nemmeno questo ha dignità a se stante nell'algebra relazionale ma è un operatore ausiliario che costituisce un mattone importante per poter implementare altri operatori. Nel caso delle GROUP BY bisogna raggruppare i valori dell'attributo, rispetto a cui raggruppo, in cluster aventi lo stesso valore e il modo più facile per farlo è quello di ordinare la relazione rispetto a quell'attributo e in automatico si ha tutti i gruppi (il raggruppamento richiede implicitamente un ordinamento). Alcune implementazioni del join (ad esempio il merge join) lavorano in modo molto efficiente su file/relazioni ordinati e quindi in certi casi può essere conveniente fare l'ordinamento e poi applicare il join. Ordinare una relazione significa restituirla tutta ordinata per un certo criterio, la restituzione avviene mediante un select.

Ordinamento ed eliminazione di duplicati, sono operatori che si applicano solo al seguito di altri (ad esempio la selezione) e sono operatori molto costosi.

- We will consider how to implement:
 - Selection (σ) Selects a subset of rows from relation.
 - Projection (π) Deletes unwanted columns from relation.
 - Join (\bowtie) Allows us to combine two relations.
 - Set-difference ($-$) Tuples in reln. 1, but not in reln. 2.
 - Union (\cup) Tuples in reln. 1 and in reln. 2.
 - Aggregation (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be **composed!** After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

Costi: non si può considerare NULLA in isolamento, l'effetto di una scelta a livello locale si ripercuote sulla continuazione e sul lavoro del resto dell'attività all'interno del quale la scelta è stata fatta. Gli operatori più costosi sono tipicamente quelli che implicano un ordinamento. Dal punto di vista del DB, il costo si esprime sempre in termini di I/O (lettura/scrittura del disco) che sono le operazioni più costose.

- L'ordinamento (il sort) è una delle operazioni più costose, il costo dell'ordinamento è $n \log n$ dove in questo caso n è il numero delle pagine (non dei record) che compongono la relazione da ordinare (quasi come se il costo all'interno di una pagina lo si possa considerare come un valore costante perché è di ordine basso e ha poco impatto sulla valutazione globale dell'algoritmo). Ogni confronto interno che viene fatto a livello di dati che sono già in memoria centrale ha costo inferiore rispetto a quelli effettuati sui dati nel disco (operazione più costosa con costo 1). Un'operazione importante (costoso perché richiede tempo) è portare una pagina in memoria centrale, quindi nonostante la complessità $n \log n$ l'ordinamento rimane particolarmente costoso.
- Il join è un altro operatore molto costoso, il costo è dell'ordine di n^2 .
- L'eliminazione dei duplicati: il costo dipende da come lo si implementa, se la relazione è già ordinata il costo è **lineare** ma se non è ordinata il costo è **quadratico** e per ogni elemento bisogna controllare a seguire se ci sono dei duplicati oppure si precede questa operazione da un ordinamento ma in questo caso il costo lo si trasferisce sul sort (quindi diventa costosa non per la ricerca dei duplicati ma per l'ordinamento che bisogna fare come prerequisito per cercare in maniera lineare il duplicato). Un altro modo per eliminare i duplicati è mettere la relazione in join con se stessa, se si fa $R \text{ join } R$ ciascun record verrà associato a tutti gli altri in cui è presente la stessa chiave quindi nel join risultante appariranno le occorrenze dei valori e da quelli si potrà facilmente eliminare i duplicati ma il join

resta costoso. Dipende quindi dalla situazione. Se si vogliono eliminare i duplicati da una selezione, che sono quindi risultati già ordinati, allora da questa selezione con una scansione lineare si possono eliminare i duplicati. Se invece la selezione non è ordinata, si procede con il sort come passo propedeutico all'eliminazione. Il costo globale può quindi basarsi su come è stato generato il risultato dalle operazioni precedenti.

Alcune volte l'ottimizzatore sceglie implementazioni più costose ma che restituiscono dati ordinati in funzione del fatto che l'ottimizzazione (all'interno di un piano della query complessivo) per cui può valere la pena pagare un po' di più un'operazione permettono un guadagno nelle operazioni successive (stima globale).

Tecniche usare nella valutazione degli operatori relazionali: tipicamente ci sono tre approcci comuni all'implementazione degli operatori. Quasi sempre si fa una combinazione delle tre strategie, raramente appaiono in maniera isolata. Ad esempio, nella maggior parte dei casi si presenta una partizione delle tre tecniche: si partiziona, si scansiona la partizione, si usa l'indice per fare certe operazioni...

- **Indicizzazione:** qualunque operatore che usa l'ordinamento può sfruttare gli indici. Ad esempio, si fare una selezione su un campo indice oppure si può fare group by usando un indice se l'indice è ordinato rispetto all'oggetto del raggruppamento.
- **Iterazione:** alcune volte è più efficiente scansionare tutte le tuple che usare gli indici. Non è sempre la più conveniente ma è quella che richiede meno prerequisiti.
- **Partizionamento:** con alcuni tipi di operazioni si può partizionare la relazione in tante sotto relazioni (ovvero frammenti della relazione stessa) per operare sulle sotto relazioni (risolvere un problema grande attraverso problemi più piccoli). Corrisponde a ridurre il problema più grande all'unione delle soluzioni di problemi parziali. Non sempre si può partizionare in maniera efficiente, i bucket invece sono un esempio che si può prestare bene al metodo.

Partizionare vuol dire: supponiamo che bisogna fare un equi join tra due relazioni, se si possiede già una tabella di hashing su questo attributo si può, anziché per ogni record della relazione esterna andare a scandire tutta la relazione interna, ci si può basare sull'hashing e quindi su blocchi di record organizzati in modo tale che record uguali stanno sullo stesso bucket e si possono limitare le operazioni di join al bucket che contiene il nome X nella prima relazione e al bucket che contiene il nome X nella seconda con la certezza che negli altri bucket non ci sia il nome e quindi non si presenti una perdita. Se si ha usato la stessa funzione di hash sullo stesso attributo nelle due relazioni, le chiavi che stanno nello stesso bucket di una relazione stanno anche nello stesso bucket dell'altra. Riduce il confronto, previsto dai join, ai soli bucket che corrispondono alle stesse chiavi e quindi agli stessi insiemi di valori.

Non sempre si può partizionare in modo ideale ma quando c'è già un partizionamento fatto in modo naturale come nel caso della bucketizzazione dell'hashing. Ci sono casi in cui si sa per certo che gli argomenti interessati di una certa relazione assumono certi valori che sono raggruppati in una precisa zona di memoria per cui si può limitare l'attenzione a quelli.

Statistiche e cataloghi: implementazioni diverse, ma equivalenti dal punto di vista del risultato (insieme di record restituiti), hanno costi computazionali a volte anche significativamente diversi riducendo drasticamente il costo sfruttando certe ipotesi/situazioni/contesti grazie all'output del risultato parziale fornito dagli operatori precedenti nel piano di valutazione della query che si sta valutando. Per scegliere le strategie applicabili ci si basa sulle strategie contenute nel **catalogo** (in modo particolare quanto sono grandi le relazioni, tipicamente quanti record ci sono per relazione, se esistono o meno degli indici, quanto sono

discriminanti i valori degli attributi, qual è la stima della cardinalità dei join che si vanno a creare, ecc.). Il catalogo contiene informazioni che in qualche modo intervengono quando l'ottimizzatore deve compiere una scelta tra le diverse strategie.

Alcune volte ci sono operazioni che sono logicamente commutative (il join) ma nonostante questo, l'ordine delle relazioni può implicare vantaggi differenti (ad esempio la scelta della relazione interna ed esterna nel join). Si parla quindi della valutazione dell'iterative looping.

Selezione semplice (Simple selection): le condizioni della selezione vengono innanzitutto convertite in CNF.

- Sequential scanning: il modo più semplice è quello di aprire il file e scandire ogni pagina per volta, ogni pagina viene portata in memoria e poi si considerano tutti i record nella pagina andando a verificare se soddisfano le condizioni, se le soddisfano allora vengono aggiunti al risultato. Per implementare questa servono almeno 2 frame di buffer (SS richiede 1 ma l'operazione non è solo SS, è SS per l'input e 1 frame per l'output). Quando il frame di output si riempie, lo si scarica su disco e lo si rende di nuovo disponibile (non è necessario mantenerlo in memoria).
- Index scan: è il metodo che sfrutta l'indice. Se la condizione è di uguaglia, si fa un accesso SH all'indice, si trova la pagina che contiene ciò che serve e la si porta nel frame del buffer per esaminarla cercando i record che soddisfano la condizione per portare questi record nel frame di output (è possibile che serva un frame di output o più). Per il tipo di operazione è sufficiente 1 frame per l'input e 1 per l'output per volta.
Se la condizione da valutare è una condizione di range, allora verrà effettuata una scansione per cercare l'intero intervallo.
 - Se l'indice è clustered (a livello di foglia non si hanno tutte le chiavi ma solo dei limitatori di intervalli che consentono di accedere all'inizio del file ordinato), si segue l'indice in modo SH, si arriva all'inizio dell'intervallo e, a questo punto, seguendo il puntatore corrispondente ci si posiziona sulla prima pagina del file ordinato dei dati (nel caso di indice clustered il file di dati è ordinato) e a partire da lì verranno individuate le operazioni all'interno del range che serviranno.
 - Se non è clustered si fa SH sull'indice e poi, essendo l'indice definito sull'attributo su cui si ha la condizione di relazione, sarà sufficiente continuare SS sulle foglie dell'indice perché le chiavi ordinate sono tutte lì (non c'è bisogno di accedere al file di dati a meno che non vengano richiesti altri dati/caratteristiche). I valori contengono tutte le chiavi dell'attributi su cui si indicizza, se bastano questi valori allora non sarà necessario accedere alle pagine dei dati stessi (ovvero al file) e si parla di **index based query**.
- Hashing: se la query ha una condizione solo per uguaglianza, si potrebbe avere un'implementazione basata sull'hashing. Se l'indice fosse un hash (e non un albero, tipicamente l'indice è hash oppure gerarchico)

Selezione complessa (Complex selection): ossia una selezione che sia una congiunzione o una disgiunzione di altre condizioni e, in particolare, si parte dall'analisi di condizioni complesse congiuntive (ovvero i record che verranno restituiti sono quelli che soddisfano l'una e l'altra condizione).

Approcci alternativi per procedere:

- si suppone di avere la condizione C1 AND C2, C1 sul cognome e C2 sulla data di nascita, si può
 - usare la condizione C1, eventualmente usare un indice se c'è, fare uno scan, arrivare ai record che soddisfano C1 e man mano che si individuano i record potenzialmente in output, si applica la condizione C2 e si verifica se bisogna andare in output davvero. Risultato: nel frame di output ci saranno i record che soddisfano sia C1 che C2

- un altro approccio è quello completamente simmetrico: se esiste un indice che permette di accedere velocemente ai dati relativi agli studenti che sono nati nell'anno X (condizione C2), da questi escludo quelli che non rispettano C1 (il cognome non rientra tra quelli di interesse)
- ipotesi di avere un indice sugli attributi di entrambe le condizioni: fa leva sul fatto che è potenzialmente costosa la scansione nella misura in cui bisogna saltare (ad esempio nell'indice clustered) e andare ad accedere alle pagine dei dati più volte. Mediante l'indice sull'attributo su cui è definita la condizione C1, si va a recuperare tutti i RID che soddisfano C1 (senza leggere i record, si accumulano solo i loro identificatori/indirizzi). Si fa a stessa cosa su C2. Dati i due insiemi di RID, i record che soddisfano entrambe le relazioni sono quelli il cui RID è stato restituito da entrambe le ricerche (quindi è presente in entrambi gli insiemi) e quindi si parla di intersezione tra i due insiemi di RID. È praticabile solo se l'indice è unclustered perché se fosse clustered non si avrebbero tutti i RID a livello di foglie.

I **fattori** che entrano in gioco nei costi: l'**accesso sequenziale**, l'**indice clustered o unclustered** (questo dipende se si ha l'accesso sequenziale o se bisogna fare i "salti"), assumendo di avere un indice (non clustered su entrambe le relazioni) entra in gioco anche la **selettività del predicato (Most selective access path)** e per stimare la selettività del predicato si usa il catalogo che permette di stimare la cardinalità del risultato. Più alta è la selettività, più è basso il numero di record restituiti (chi è più selettivo pesca meno) e scegliere il criterio più selettivo ha il vantaggio che limita a un numero di record inferiore il fatto di dover essere portato in memoria per andare a testare l'altro criterio. Dovendo fare C1 e C2, il criterio più conveniente è quello più selettivo. Un altro fattore è relativo all'eventuale richiesta di ordinamento dei risultati, l'uso dell'indice potrebbe essere un aiuto ad avere un risultato già ordinato se uso come primo criterio quello rispetto a cui devo ordinare (in questo si è più facilitati ad avere un risultato ordinato perché sono stati recuperati i record ordinatamente dall'indice andando a sfruttare il fatto che l'indice è già ordinato di suo (**ordinamento su un attributo indice**). Stessa cosa se l'utente richiede delle clausole che implicitamente presuppongono un ordinamento.

Aspetti diversi da tenere in conto per scegliere le diverse alternative: situazione del catalogo, statistiche sulla presenza dei dati nelle relazioni, requisiti dell'utente in termini di risultati esplicitamente ordinati o no ma con richieste come DISTINCT e GROUP BY che a loro volta presuppongono un ordinamento.

Proiezione (Projection): è un operatore che ricevendo in input una relazione con un certo schema (quindi un certo insieme di attributi) restituisce per ciascun record, della relazione stessa, un nuovo record che contiene soltanto i campi specificati nella condizione di proiezione per cui la cardinalità della relazione in uscita è la stessa della cardinalità della relazione a cui la si applica a meno che ci sia la clausola DISTINCT o WHERE, senza di queste il numero di pagine coincide a quello della tabella.

Implementazione: si prevede che venga aperto il file, si scandiscono tutte le pagine e in modo particolare leggendo a una a una le pagine del file (SS) per ogni pagina si fa fetching delle tuple della pagina stessa, si identificano i campi da proiettare e li si inseriscono all'interno del frame di output. Anche nella proiezione (implementata in modo basic, ovvero naive) sono sufficienti 2 frame di buffer (uno per la lettura e uno per l'output).

- È utile usare gli indici nei casi di **index only query** se gli indici definiti sulla relazione sono definiti sugli attributi da proiettare e sono di tipo non clustered.
- L'indice è utilizzabile solo se è un indice non clustered (con quelli clustered le foglie non contengono tutte le chiavi ma solo un sottoinsieme che aiuta a identificare l'inizio di un range nel file dei dati a cui poi bisognerà accedere).

- Se servono i dati ordinati e si ha un indice a disposizione, è possibile di nuovo sfruttare un indice perché in questo caso l'ordinamento diventerebbe di nuovo molto naturale. Si può fare anche le l'indice è clustered perché si vanno comunque a trovare le foglie divise per range e poi si possono leggere i file in maniera ordinata e quindi il risultato sarà ordinato (naturalmente questo caso implica la lettura di file).
- Con la clausola DISTINCT può tornare utile sia un indice gerarchico che hash. Anche hash perché se si ha la clausola DISTINCT ma non una richiesta di ordinamento, si sa comunque che le eventuali coppie multiple di una stessa chiave sono nello stesso bucket nel caso dell'hashing quindi si può accedere bucket per bucket ed eliminare le versioni diverse della stessa chiave accedendo alla singola pagina del singolo bucket dell'hashing (non è necessario scandire l'intero file per cercare la chiave, costo quadratico, perché le chiavi saranno tutte nello stesso bucket).
- L'eliminazione dei duplicati è costosa quindi avviene solo quando l'utente specifica la clausola DISTINCT.

Ordinamento (Sort): il quick-sort è un algoritmo molto efficiente con complessità $n \log n$ che non riesce altro spazio oltre ai dati che deve ordinare (dimensione contenuta) ma lo si può applicare solo nei casi in cui la relazione è sufficientemente piccola da stare in memoria, in caso contrario NO. Nel caso tipico, ad esempio quando si fa il bulk loading di un B+ tree bisogna ordinare intere relazioni che sono ben più grande dello spazio nel buffer e quindi gli algoritmi come il quick-sort diventano improponibili perché uno swap tra elementi in memoria costa pochissimo ma se per farlo si causano molti page fault, le operazioni di I/O diventano pesantissime. Lavorare in-site (sul posto) è molto pesante perché implica molte operazioni I/O.

Il quick-sort lavora in-place, questo vuol dire: dato un array da ordinare, non richiede alcun spazio aggiuntivo ma effettua l'ordinamento ricorsivamente effettuando swap di elementi ordinando passo dopo passo l'array fino a raggiungere la condizione di terminazione quando è tutto ordinato. Localmente si fa un ordinamento e poi si scambiano via via gli elementi che devono essere scambiati fino al completo ordinamento. Ricorsivamente serve avere in memoria tutto l'array da ordinare, siccome l'ipotesi di lavoro è che non ci stia tutta la relazione ordinata in memoria questo vuole dire che si presenterebbero moltissimi page fault. Il fatto di lavorare direttamente sui dati che deve ordinare, vuol dire tornare più volte sulle stesse pagine e quindi fare tante operazioni I/O che appesantiscono in modo significativo il costo dell'algoritmo stesso.

Altri algoritmi come Heap-sort e Merge-sort NON sono in-place, in particolare il merge-sort (che è quello più utilizzato per il sort esterno) è fatto in modo tale da richiedere il doppio dello spazio (nei db lo spazio è potenzialmente un problema), sembra che sia peggio ma in realtà non è così grazie alla natura degli accessi che si fanno all'array da ordinare sono via via locali a differenza del quick-sort. Passo dopo passo si formano delle piccole porzioni di array ordinati ogni volta.

Metodi di ordinamento: il quick sort è uno tra i più efficienti ma diventa improponibile e pesante perché un ordinamento di questo genere della relazione non ci sta nel buffer manager e quindi si richiederebbero troppe operazioni I/O. Il quick sort lavora in place, dato un array da ordinare non richiede spazio aggiuntivo ma effettua l'ordinamento effettuando gli swap degli elementi portando via a via l'array a essere sempre più ordinato. Avviene un ordinamento locale fino ad arrivare all'ordinamento dell'intero array. Il fatto di operare direttamente sui dati che comporta il saltare avanti e indietro, quindi vengono effettuati tanti swap che comporta appesantimenti. (da approfondire)

Merge-sort ed external-sort sono fatti per non lavorare in-place. Il merge-sort richiede dello spazio aggiuntivo ma con opportuni accorgimenti, permette di sfruttare bene il buffer. External-sort è il sort di una relazione che non è completamente interno in memoria, può funzionare anche solo con 3 pagine del buffer. Verranno

quindi ordinate coppie di pagine contigue (due pagine per l'input e una per l'output), ad ogni iterazione raddoppia la lunghezza (ovvero il numero di pagine nei runs) e il file si presenta sempre più ordinato. Pagina per pagina la si ordina al suo interno.

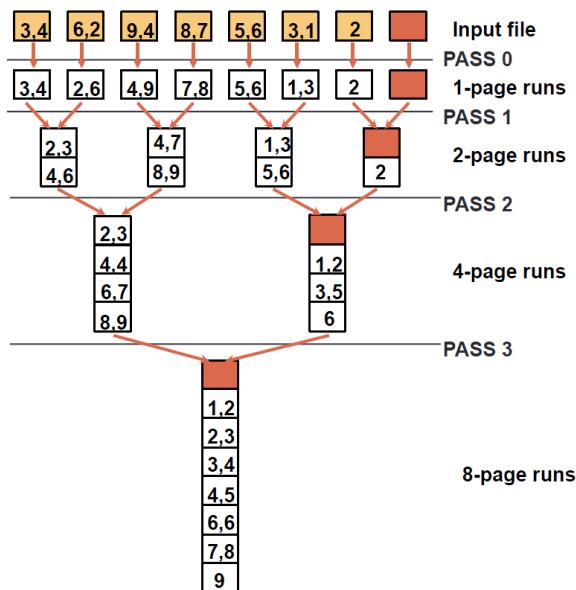
Nell'external sort sono $2N$ (una per le letture e una per le scritture).

Nota: approfondire il concetto "run".

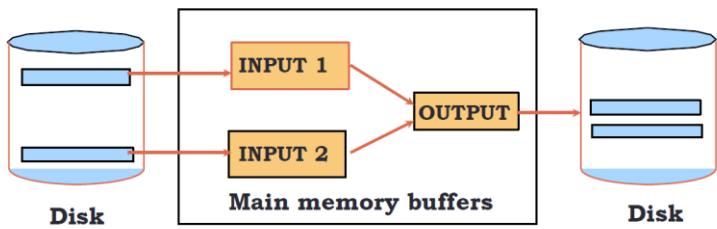
Tre soli frame di buffer possono bastare per poter implementare l'external-sort. Partire con run più lunghi comporta la riduzione del numero di livelli per arrivare alla soluzione ordinata (riduzione del numero di iterazioni per arrivare all'ordinamento globale). Con più frame si otterrebbero più run ordinati in minor tempo.

Merge-sort: alcuni accorgimenti permettono di utilizzare opportunamente il buffer a disposizione. Quello che viene chiamato 2-way-sort è questo tipo di operazione di merge-sort che permette l'implementazione dell'external-sort (sort di una relazione non completamente contenuta in memoria, contrapposizione al quick-sort che è un sort interno che per funzionare in maniera efficiente presuppone la presenza in RAM dell'intera relazione). È possibile implementare un external-sort basato su merge-sort utilizzando soltanto 3 frame del buffer. Tre perché due servono per leggere i flussi di dati da andare a fondere per fare il merge e il terzo serve per andare ad accumulare l'output in cui si va a costruire la "fusione" dei due flussi che via via si accumulano. A ogni iterazione raddoppia la grandezza della porzione di file ordinata.

Con un file di 5M pagine, si può pensare inizialmente di ordinare coppie di pagine contigue (ex le prime due). Le pagine al loro interno sono ordinate e prendono il nome "run" (un "sotto file", una porzione di file). Ad ogni iterazione il file raddoppiera le porzioni ordinate. Le porzioni ordinate verranno raddoppiate $\log_2(n)$ volte prima di arrivare al file completamente ordinato, dove n è il numero delle pagine.



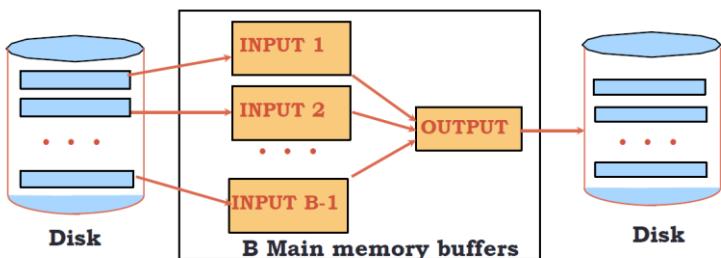
Nota: naturalmente la dimensione di queste pagine non è realistica ed è al solo scopo didattico.



3 frame di buffer sono sufficienti per implementare il Two-Way External Merge Sort anche per file di 5k pagine (non è importante la dimensione, cambia solo il numero di passi e quindi la profondità dell'albero). Se avessi più di 3 frame (ci sono strategie di gestione del buffer che consentono di assegnare più frame di buffer a una stessa operazione) si possono portare più pagine di buffer in memoria contemporaneamente generando immediatamente di run più lunghi. Il vantaggio di partire con run più lunghi vuol dire ridurre il numero di passaggi per raggiungere il file ordinato (riduzione del numero di iterazioni). Vengono generati meno run ordinati ma più lunghi.

In questo caso, se N è il numero di pagine del file da ordinare e se ciascun gruppo del file da ordinare contiene B pagine, si producono N/B run ordinati ciascuno composto da B pagine. B è il numero di frame di buffer che si possono utilizzare per effettuare le operazioni, $B-1$ run su cui fare il merge e 1 per l'output.

A differenza del quick sort che va a modificare le pagine in-site, il merge sort prevede una copia in un frame di output quindi se vengono utilizzati B frame, $B-1$ sono per l'input e 1 per l'output. Fondendo $B-1$ run per volta, si generano dei run ordinati di lunghezza $B-1$.



Costi:

- Numero di iterazioni (ovvero di passi) = $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Costo: $2N * (\# \text{ di passi})$

E.g., with 5 buffer pages, to sort 108 page file:

- Pass 0: $\lceil 108 / 5 \rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages)
- Pass 1: $\lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages)
- Pass 2: 2 sorted runs, 80 pages and 28 pages
- Pass 3: Sorted file of 108 pages

Nota: $4 = B-1 = 5-1$, dove $B=5$.

Number of Passes of External Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Internal Sort Algorithm: anziché leggere i dati dal file iniziale nel modo in cui sono presenti, li si possono leggere e inserire in una **heap** (ovvero una coda con priorità) in cui si tengono i dati ordinati in modo tale che con costo costante si riesce ad estrarre il minimo data la natura dell'organizzazione dei dati all'interno della struttura. L'idea è: anziché prende un blocco di dati, prendili e organizzarli in una heap, dopo di che utilizza una heap per andare a popolare quello che sarà un frame di output, non utilizza quindi un quick-sort ma una heap. Il vantaggio di questa strategia è che man mano che estrae un elemento dalla heap, si valuta se si può inserirne un altro a posto (mantiene la heap piena) con l'obiettivo di creare immediatamente (in questo primo passo) dei run più lunghi.

Nel primo passo di portano in memoria B blocchi e si crea una coda di priorità dopo di che se ne estrae uno e lo si inserisce nell'heap.

- Nel caso migliore si crea direttamente un ramo lungo N e la situazione è già ordinata, accade solo se il file è già ordinato.
- Caso medio: mediamente si creano run lunghi $2B$.
- Caso peggiore: si fa tutto il lavoro per avere solo run di lunghezza B (in questo caso era meglio usare il quick-sort).

Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- An alternative is “tournament sort” (a.k.a. “heapsort”)
 - **Top:** Read in B blocks
 - **Output:** move smallest record to output buffer
 - Read in a new record r
 - insert r into “heap”
 - if r not smallest, then **GOTO Output**
 - else remove r from “heap”
 - output “heap” in order; **GOTO Top**

Join: è un operatore derivato dal prodotto cartesiano e una selezione, è definito come una selezione rispetto alla condizione di join su un cartesiano (molto costoso). Seleziona (dal prodotto cartesiano di due relazioni)

solo quelle coppie di record che soddisfano la condizione di join, non è un operatore kernel ma derivato. Il prodotto cartesiano è molto costoso, applicarci una selezione sopra aumenta i costi.

Verrà vista l'implementazione del join come operatore a se stante, ovvero non visto come un operatore derivato che applica la selezione al prodotto cartesiano ma visto come un operatore che opportunamente combina la fase di abbinamento delle coppie di record e la fase di verifica del soddisfacimento della condizione in modo tale da sperare di ottenere un costo computazionale inferiore a quello che si avrebbe semplicemente applicando la selezione sul cartesiano.

Equality Joins With One Join Column

```
SELECT *
FROM Reserves R1, Sailors S1
WHERE R1.sid=S1.sid
```

- In algebra: $R \bowtie S$. Common! Must be carefully optimized.
- $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.
- Assume: M tuples in R, p_R tuples per page, N tuples in S, p_S tuples per page.
 - In our examples, R is Reserves and S is Sailors.
- We will consider more complex join conditions later.
- *Cost metric*: # of I/Os. We will ignore output costs.

Ipotesi di lavoro: si hanno le due relazioni R e S con cardinalità eventualmente diverse, non ci sono requisiti in merito alla cardinalità della prima e della seconda, non ci sono nemmeno requisiti che legano a priori il numero di record che stanno in una stessa pagina dell'una e dell'altra. Per valutare i costi è importante sapere quante sono le pagine di un operando e dell'altro, quanti sono i record per pagina all'interno di una relazione e dell'altra

Simple Nested Loops Join: è il join più semplice, che è l'implementazione a cui è stato fatto riferimento, ad esempio, quando si è parlato di gestione del buffer, politiche di rimpiazzamento e quando si parlava di sequential flooding se si applica una politica di rimpiazzamento LRU perché ci sono dei pattern di accesso sequential looping che fanno sì che si debba tornare più volte sulla stessa pagina.

Simple Nested Loops Join prevede che si legga ciascun record di una delle due relazioni (del primo operando). Il primo operando è la relazione esterna e il secondo è la relazione interna. Per ogni record della relazione esterna, bisogna andare a considerare tutti i record della relazione interna e verificare se il record che si sta considerando soddisfa la condizione di join e in questo caso lo manda in output.

Nota:

- è ancora molto usato soprattutto per le sue generalità, non richiede ordinamenti, non richiede alcun requisito sulla condizione di join che si sta testando, è molto facile, semplice da implementare e ha caratteri di grandissima generalità, quindi non è soggetto a ipotesi di lavoro.
- **NON è un algoritmo “bloccante” (blocking):** vuol dire che se si sta facendo join tra due relazioni che a loro volta sono generate da altre operazioni, se la si vuole fare orientata alla tupla si può pensare di dire che man mano che vengono generati frammenti dei risultati parziali da combinare, comincio a vedere se fanno match. Ovvero, possono iniziare a valutare il join prima che sia completamente

completata la generazione dei risultati parziali da combinare. Man mano che si hanno dei record prodotti si controlla se fanno match. Ovvero, si può operare man mano che i risultati vengono prodotti.

Simple Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if  $r_i == s_j$  then add  $\langle r, s \rangle$  to result
```

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
 - Cost: $M + p_R * M * N = 1000 + 100*1000*500$ I/Os.
- Page-oriented Nested Loops join: For each page of R, get each page of S, and write out matching pairs of tuples $\langle r, s \rangle$, where r is in R-page and S is in S-page.
 - Cost: $M + M*N = 1000 + 1000*500$
 - If smaller relation (S) is outer, cost = $500 + 500*1000$

Nota:

- p_r è il numero di record per pagina.
- In termini di costi, l'operatore join NON è commutativo perché cambia la complessità in base dalla cardinalità. L'ottimizzatore sceglierà la relazione con cardinalità inferiore come relazione interna perché è quella che verrà letta più volte.
- C'è una ripetizione (ridondanza) perché la relazione interna viene letta troppe volte.
- nel primo caso (**tuple**) conviene che sia minore il numero di pagine della relazione interna, nel secondo caso (**paginae**) invece è conveniente avere esterna la relazione con cardinalità inferiore.
- l'equi-join può avere vantaggi nel caso in cui la relazione interna sia ordinata perché così non necessariamente bisogna leggerla tutta (sort-merge join). Se i valori sono tutti distinti, ognuno viene letto una volta sola

Index Nested Loops Join: sfrutta il fatto di sapere di avere un indice definito sulla relazione, per cui se si ha un indice si può sfruttare l'indice stesso. Si può quindi partire dalla relazione esterna e per ogni record della relazione esterna, si può sfruttare il fatto che sulla relazione interna è definito l'indice sull'attributo su cui interessa cercare il match e andare a vedere se quel valore con cui si sta cercando il match è presente nella relazione interna. Quindi scandisce con un'unica lettura la relazione esterna, anziché fare tutta la scansione della relazione interna per vedere record per record se fanno match, si sfrutta l'indice che è presente sulla relazione interna e si va a cercare i match. Ciascun record della relazione esterna comporta un accesso all'indice per andare a cercare il corrispondente elemento nella relazione interna, quindi il costo di quest'operazione è fortemente condizionato dal numero di volte con cui bisogna accedere all'indice e dal costo dell'accesso all'indice stesso (anche clustered e non clustered influisce).

Index Nested Loops Join

```

foreach tuple r in R do
    foreach tuple s in S where  $r_i == s_j$  do
        add  $\langle r, s \rangle$  to result

```

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
 - Cost: $M + (M * p_R) * \text{cost of finding matching S tuples}$
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
 - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
 - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
 - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.
- Hash-index (Alt. 2) on *sid* of Reserves (as inner):
 - Scan Sailors: 500 page I/Os, 80*500 tuples.
 - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.

Block Nested Loops Join: si è passati dall'agire a livello di record, poi a livello di pagine e ora ci si chiede se si può estendere questo tipo di vantaggio che si ha avuto passando dal record alla pagine se si può contare su un numero di frame nel buffer tale da consentire di portare in memoria più pagine simultaneamente della relazione esterna, allora si può proporre il block nested loop join. Invece che lavorare a livello di pagine, si lavora a livello di blocchi.

Dato un numero B di pagine del buffer ($B+1$ perché 1 è sempre per l'output), si possono tenere un numero di pagine minore di $B-1$ (quindi $\leq B-2$) per la relazione esterna. Le rimanenti pagine per quella interna e un frame per la pagina da mandare in output. Si estende quindi a livello di blocco in concetto che precedentemente è stato fatto a livello di singola pagina, ossia anziché portare in memoria una pagina per volta della relazione esterna e quindi ciclare completamente sulla relazione interna tante volte quante sono le pagine. Si possono portare in memoria dei blocchi di $B-2$ pagine e questo porta a ciclare sulla relazione interna un numero di volte decisamente inferiore, ossia un numero di volte pari al numero di blocchi della dimensione che si è riusciti a formare.

Se si ha un file di 80 pagine e se ne riescono a portare in memoria simultaneamente 10, allora si riuscirà a ciclare sulla relazione interna soltanto $80/10 = 8$ volte e questo consente di ottimizzare ancora l'algoritmo e quindi la valutazione del join perché si ciclerà molto meno la relazione interna.

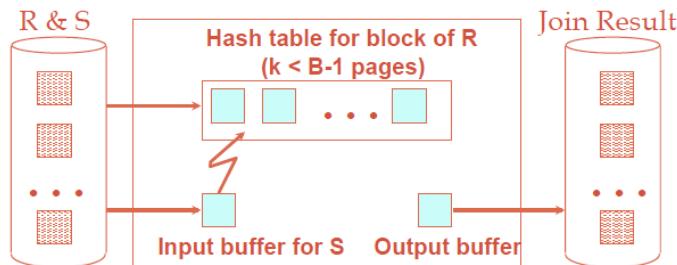
- **Primo caso:** vengono caricate in memoria $k < B-1$ pagine di R e 1 di S, quindi 1 frame viene riservato all'output. L'effetto è che il numero di volte in cui devo ciclare sulla relazione interna si riduce

drasticamente di un fattore pari alla dimensione B-2, questo perché ciascuna pagina della relazione interna si riesce a confrontare con tutte quelle del blocco che si sta considerando.

- **Secondo caso:** vengono caricate in memoria $k < B-1$ pagine di S e 1 di R, quindi 1 frame viene riservato all'output. Il vantaggio di questa seconda strategia è che nonostante bisogna comunque fare tanti cicli e avendo a disposizione tanti frame di buffer per la relazione interna, si riesce a riutilizzare con politica MRU pagine già in memoria e quindi, è vero che si cicla sulla pagina ma quello che non succede è che **ogni** lettura su una pagina determina una lettura da disco quindi il fatto di ciclarci non diventa necessariamente un'operazione costosa perché si lavora direttamente in memoria centrale limitando i page fault ai casi in cui la politica MRU porta a fare il rimpiazzamento dovuto.
- **Quello che si fa tipicamente è:** se si hanno più frame di buffer si cerca una soluzione di compromesso, ad esempio metà a R e metà a S. Dando $B/2$ a R si riducono il numero di cicli su S, dando $B/2$ a S si riducono i page fault che bisogna fare per ciascun ciclo perché poi con la politica MRU si riesce a gestire il tutto. Tipicamente si concede quindi metà a testa.

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ‘‘block’’ of outer R.
 - For each matching tuple r in R-block, s in S-page, add $\langle r, s \rangle$ to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
 - #outer blocks = $\lceil \# \text{ of pages of outer} / \text{blocksize} \rceil$
- With Reserves (R) as outer, and 100 pages of R:
 - Cost of scanning R is 1000 I/Os; a total of 10 blocks.
 - Per block of R, we scan Sailors (S); 10*500 I/Os.
 - If space for just 90 pages of R, we would scan S 12 times.
- With 100-page block of Sailors as outer:
 - Cost of scanning S is 500 I/Os; a total of 5 blocks.
 - Per block of S, we scan Reserves; 5*1000 I/Os.
- With sequential reads considered, analysis changes: may be best to divide buffers evenly between R and S.

Sort-Merge Join (R ><i=j S): funziona solo con join per uguaglianza.

Le tecniche viste fino a questo punto non fanno riferimento al possibile ordinamento delle operazioni su cui si sta operando e non fanno ipotesi sul tipo di condizione che si va a testare (condizioni di uguaglianza, disuguaglianza, ecc.), sono tecniche molto generali.

È un algoritmo “**bloccante**” mentre, ad esempio, il Nested Loop Join non lo è. Bloccante vuol dire che non si può fare il match di una relazione prima che la si “conosca” completamente.

“**blocking**”: ad esempio nel caso del Sort-Merge Join fatto su dati generati in modalità stream senza la garanzia che vengano generati ordinati, quindi se l’algoritmo deve essere proceduto da un’operazione di ordinamento allora io sono **bloccato** e non posso cominciare perché non ho ancora finito ordinamento e si può ordinare soltanto quando ho tutto. In questo senso, questo tipo di implementazione è “blocking” nel senso che non può cominciare se non dispone già completamente dei suoi operandi perché deve poterseli ordinare, mentre un join che considera i record indipendentemente può già operare con risultati parziali (ad esempio il Nested Loops Join).

NB: il blocking è una debolezza.

Quindi il vantaggio di quello orientato alle tuple (piuttosto che alla pagina o al blocco) è quello di non essere bloccante mentre tutti gli operatori che presuppongono un input ordinato sono potenzialmente bloccanti perché l’attività di ordinamento dell’input presuppone che gli interi argomenti su cui bisogna operare siano a disposizione.

Example of Sort-Merge Join

			<u>sid</u>	<u>bid</u>	<u>day</u>	<u>rname</u>
<u>sid</u>	<u>sname</u>	<u>rating</u>				
22	dustin	7	45.0	28	103	12/4/96
28	yuppy	9	35.0	28	103	11/3/96
31	lubber	8	55.5	31	101	10/10/96
44	guppy	5	35.0	31	102	10/12/96
58	rusty	10	35.0	31	101	10/11/96
				58	103	11/12/96
						dustin

- Cost: $M \log M + N \log N + (M+N)$
 - The cost of scanning, $M+N$, could be $M \cdot N$ (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

Nel caso specifico dell’equi-join, quindi di un join per uguaglianza, si possono ottenere dei grossi vantaggi se si sfrutta l’informazione relativa al fatto che R e S sono ordinate o se non lo sono potrebbe risultare conveniente ordinarle per poi sfruttarle durante il join. Questa strategia può migliorare MA non vuol dire che migliorerà, in particolare migliora di molto se la condizione di join è molto selettiva (in particolare se si fa il join tra due chiavi per cui ciascun elemento della prima relazione corrisponde ad al più un elemento della seconda relazione, allora il miglioramento può essere davvero notevole perché se le relazioni sono già ordinate di riesce ad ottenere il join con un’unica lettura della prima e della seconda relazione). Se la condizione di join non è selettiva e nel caso peggiore tutti i record della prima fanno match con tutti i record della seconda, allora questa strategia diventa estremamente sconveniente per la complessità diventa comunque pari a quella del prodotto cartesiano MA si aggiunge anche l’eventuale costo dell’ordinamento (non migliora, anzi peggiora).

Un esempio di situazione in cui tutto fa match con tutto è quanto tutti gli elementi della prima relazione sono uguali e tutti gli elementi della seconda solo a loro volta gli stessi, per cui ciascun elemento della prima

relazione deve esser messo in corrispondenza con tutti quelli della seconda. È un caso degenero, molto raro. Anche senza arrivare a questo caso così estremo, rimane una situazione pesante e il vantaggio continua a essere molto contenuto.

Quando si scandiscono le relazioni da combinare (equi join), non si hanno relazioni ordinate per cui per ogni record della relazione esterna bisogna scandire tutta la relazione interna perché dopo aver trovato un eventuale match non si può concludere se ce ne saranno o meno altri, per arrivare a questa conclusione che non ci saranno altri record che faranno match con quello della relazione esterna selezionato bisogna aver scandito tutta quanta la relazione interna (questo perché la relazione interna non è ordinata quindi gli elementi che fanno match per uguaglianza non sono necessariamente contigui). Se io già sapessi che la relazione interna è ordinata e non sapessi niente nella relazione interna, si avrebbe già un beneficio perché si ridurrebbero le letture nella relazione interna. Tuttavia, l'ordinamento sarebbe utile alle chiavi che stanno all'inizio dell'ordinamento ma non a quelle alla fine perché comporterebbe comunque la lettura dell'intera relazione interna.

Se fossero ordinate tutte e due, si può portare in memoria centrale la prima pagina della relazione interna e di quella esterna e avanzano con il puntatore esattamente come si fa col merge. Considero il valore che c'è sulla relazione esterna e quello sulla relazione interna, se sono uguali si ha un caso di match (e quindi va in output) e in caso contrario il minimo dei due avanza (perché sono ordinati) e se il minimo non sta facendo match con quello che c'è nell'altra relazione vuol dire che quel minimo non ha più speranza di fare match andando avanti. Quindi, se i valori sono tutti distinti, ciascuna relazione viene letta una volta sola, se invece non sono tutti i distinti bisogna decidere quale delle due relazioni è da trattare da relazione esterna e quale da relazione interna (bisogna decidere quale dei due valori fermare e procedere con i confronti fino a quando non si ottiene il valore successivo, è come se si avesse un prodotto cartesiano all'interno dei due "blocchetti" che fanno match). Con i valori tutti distinti la situazione è completamente simmetrica.

Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
 - With $B > \sqrt{L}$, where L is the size of the larger relation, using the sorting refinement that produces runs of length $2B$ in Pass 0, #runs of each relation is $< B/2$.
 - Allocate 1 page per run of each relation, and 'merge' while checking the join condition.
 - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
 - In example, cost goes down from 7500 to 4500 I/Os.
- In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.

Sort-Merge Join richiede il sort a monte prima della fase di matching, per fare sort-merge fa prima sort delle relazioni e poi match delle due relazioni ordinate. Il merge fatto ricorda tanto l'operazione di merge-sort (fusione delle due relazioni visto con il sort esterno, i due puntatori scorrono le due relazioni ordinate e copiano in output solo i match).

È possibile combinare (simultaneamente) la fase di ordinamento del file applicando merge-sort eventualmente (questo perché è quello solitamente utilizzato per il sort esterno) e l'applicazione del join?

- L'obiettivo è integrare la fase di matching con quella dell'ordinamento.
- Si potrebbe scegliere di proseguire nel seguente modo: quando inizio a considerare R per ordinarla, mi fermo al primo passo dell'ordinamento (ossia il passo ne merge-sort in cui era stato generato tanti

run ordinati, ovvero tante sotto sequenze di file ordinate), stessa cosa per S. A questo punto, anziché fare il merge-sort per fonderli, posso pensare di far partire direttamente il join. Porterei in memoria principale una pagina per ciascuno dei run che è stato creato (ovviamente serve più di un frame di buffer in memoria).

- Supponiamo di avere un file di 20 pagine e di essere riusciti, con l'ordinamento, a creare run lunghi 5 pagine ciascuno e supponiamo di avere 5 frame di buffer (5 va bene perché è maggiore di 2 che è il minimo) a disposizione per ciascuna delle relazioni (le relazioni sono lunghe 20, 5pgine * (5frame - 1output)). Creo run che non sono più di 5 per ciascuna relazione, ovvero il numero di run creato è tale per essere contenuto in memoria centrale nel buffer.
- A questo punto inizia il join esattamente nel modo in cui sarebbe iniziato il merge nella versione orientata ai blocchi, ossia leggo in memoria una pagina per ciascun run e comincio a fare le operazioni di join.
- Fino a questo punto ho fatto un passo di merge-sort è costato $2M$ (oppure $2N$ a seconda della valutazione che è stata effettuata, il costo è dato dalla lettura di ciascuna pagina portandola nei frame di buffer e dalla scrittura dopo averle ordinate).
- Dopo si passa all'operazione di fusione che si fa sfruttando l'esistenza di più pagine simultaneamente presenti nel buffer e ordinate. Quindi prendo la prima pagina di una relazione e confronto tutti i record con i record presenti nelle prime pagine dell'altra. Quando avrò completato l'analisi della prima pagina (avendo fatto il matching con tutte le pagine dell'altra), potrò sovrascriverla con la seconda pagina dello stesso run. Nel caso migliore si riduce a un'unica lettura dei frame, nel caso peggiore (questo deriva dal fatto che c'è una selettività bassa del predicato di selezione) il costo sarà $M*N$. Si riesce a risparmiare nei costi perché sono stati fatti dei run sufficiente lunghi da poter operare in locale in memoria centrale.
- Idea alla base: tutti i valori più piccoli tra quelli presenti nelle relazioni saranno in testa ai rispettivi run creati, quindi il fatto di avere in memoria centrale tutte le prime pagine dei diversi run creati massimizza la possibilità di avere matching tra i valori più piccoli di una relazione e quelli dell'altra perché sono tutti finiti nelle prime pagine dei diversi run.
- Il matching ha quindi la stessa logica dell'ordinamento che è quello di scandire run ordinati alla ricerca di match. Quindi anziché leggere i blocchi come facevo nel merge sort per creare una relazione ordinata in cui tutti sono contenuti, leggo i blocchi per creare una relazione di output in cui chi non ha un corrispondente tra gli altri va escluso mentre si salva sull'output solo quelli per cui trovo un valore corrispondente. Si fa con la stessa logica del merge-sort, solo che quelli che non hanno il corrispondente, anziché finire nella loro posizione corretta (come sarebbe previsto dall'algoritmo di ordinamento) questa volta vengono semplicemente scartati. Dopo che vengono consumati tutti i minimi, si fanno avanzare i puntatori.

Il merge sort prende il minimo di tutto e lo mette in output, ora invece viene preso il minimo di tutte le pagine della relazione R e lo confronta con il minimo di tutte le pagine della relazione S, se i minimi sono uguali vuol dire che fanno match e quindi si creano tutti i possibili "incroci" tra i minimi (possono essere su un'istanza sola oppure su più istanze). Se i minimi della prima relazione sono minori rispetto ai minimi della seconda, si può concludere che (siccome i run sono ordinati) il minimo della prima relazione non abbia un corrispondente valore nell'altra e quindi si può portare avanti il puntatore della prima relazione (quindi proseguono i confronti e gli avanzamenti dei puntatori della prima e della seconda relazione). Naturalmente se il minimo della prima relazione è maggiore della seconda, si trae la stessa conclusione e avanza i puntatore di S. Viene quindi effettuata una scansione in avanti di una e dell'altra relazione.

In sostanza, ogni volta vengono confrontati i minimi per escludere la possibilità di join o per valutarla per mandare i record in output.

Attenzione: nel caso del merge-join che prosegue in maniera simmetrica non ha una relazione esterna e una interna proprio perché cade questo concetto al seguito della navigazione simmetrica delle relazioni.

Note generali:

- Ci sono situazioni dove l'input è già ordinato.
- Ci sono situazioni in cui i singoli operatori logici dell'algebra corrispondono a diversi operatori fisici.

Hash Join: lo si può applicare solo negli equi-join, ossia quando la condizione di join è di uguaglianza, questo per la natura intrinseca del concetto di hash. L'idea dell'hash join per un certo senso è molto simile a quello del Blocked Nested Loops Join.

Nell'hash join si continuano a scandire entrambe le relazioni, alla prima passata si creano le partizioni di dati da portare su disco sulla base della funzione di hash. Quindi date le relazioni indicizzate mediante hash, anziché fare l'ordinamento col sort-merge join (lettura di tutte le pagine e scrittura in memoria sulla base dell'ordinamento) faccio una passata per cui leggo tutte le pagine e le scrivo in memoria ma riorganizzate in modo tale che i record, che corrispondono allo stesso bucket, siano sulla stessa pagina di disco. Posso pensare di organizzare i dati in pagine diverse in base al valore dell'hashing, questo è quello che succede dei bucket stessi dell'hashing. I bucket sono zone di memoria separate, ciascuna dei quali ha dimensione 1 pagina che è anche la dimensione di 1 frame di memoria.

L'indice di hash è sulle chiavi che danno il puntatore, quando si fa i join bisogna restituire i record interi. Ci sono due strategie:

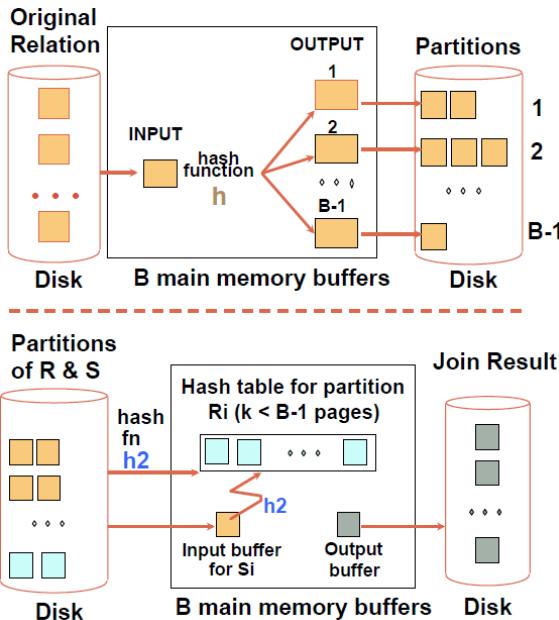
- Creare delle partizioni a priori guidate dall'indice di hashing.
- Fare il lavoro del join direttamente sugli indici ma restituendo anche i relativi puntatori oltre che le chiavi in modo tale da poter recuperare tutti i record completi.

Hash join non è simmetrico perché porta in memoria blocchi di frame della relazione esterna e con quelli faccio eventualmente più passate sul blocco corrispondente della relazione interna. Esistono però anche delle versioni non blocking e simmetriche secondo certi accorgimenti. È più conveniente la simmetria per togliere la decisione iniziale in merito a quale relazione deve essere interna e quale esterna.

Nota: guardare sul libro.

Hash-Join

- Partition both relations using hash fn h : R tuples in partition i will only match S tuples in partition i .



Observations on Hash-Join

- #partitions $k \leq B-1$ (why?), and $B-2 >$ size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k , we get:
 - $k = B-1$, and $M/(B-1) < B-2$, i.e., B must be $> \sqrt{M}$
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join

- In partitioning phase, read+write both relns; $2(M+N)$. In matching phase, read both relns; $M+N$ I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
 - Given a minimum amount of memory (*what is this, for each?*) both have a cost of $3(M+N)$ I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
 - Sort-Merge less sensitive to data skew; result is sorted.

Attenzione:

General Join Conditions

- Equalities over several attributes (e.g., $R.sid=S.sid$ AND $R.rname=S.sname$):
 - For Index NL, build index on $<sid, sname>$ (if S is inner); or use existing indexes on sid or $sname$.
 - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
- Inequality conditions (e.g., $R.rname < S.sname$):
 - For Index NL, need (clustered!) B+ tree index.
 - Range probes on inner; # matches likely to be much higher than for equality joins.
 - Hash Join, Sort Merge Join not applicable.
 - Block NL quite likely to be the best join method here.

Query Optimization

- Query languages are declarative
- We need to
 - convert declarative statements into executable statements
 - (enumerate query plans)
 - estimate the cost of the executable statements
 - choose the cheapest executable statement among all alternatives
(search in the query plans space)

Il modulo di ottimizzazione delle query è molto utile perché, com'è stato visto nei DB relazionali la query è descritta nel calcolo relazionale ed è un linguaggio dichiarativo (ovvero stabilisce cosa si vuole ma non come ottenerlo), è a carico del DBMS convertire uno statement dichiarativo nel linguaggio dell'algebra relazionale. È possibile che la stessa query dichiarativa corrisponda a diversi statement dell'algebra relazionale e a ogni operatore corrispondono diverse implementazioni diverse ognuna con la sua particolarità. Ciascuna istanza di DBMS ha a disposizione molte implementazioni dello stesso operatore e query per query valuta l'implementazione ideale a seconda della situazione. Scegliere l'operazione ottima in uno specifico momento comporterebbe un costo superiore all'eventuale vantaggio che questa scelta comporterebbe, quindi si rinuncia l'opportunità di scegliere l'ottimo e si lavora soprattutto su euristiche per cercare di evitare/escludere le scelte peggiori (situazione di compromesso tra il costo della scelta e il vantaggio che la scelta comporta).

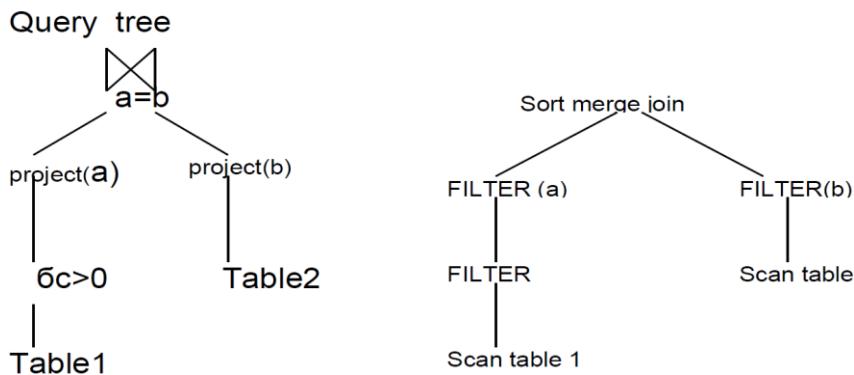
Cosa fa il DBMS? Dato uno statement di una query che l'utente ha formulato, il DBMS deve trovare dei piani fisici per la query e quindi deve trovare una riscrittura procedurale della query indicando per ciascun operatore l'implementazione fisica dello stesso che è prevista. In linea di principio, si può pensare che ci si muova come un'ottimizzazione all'interno di uno spazio di ricerca che è costituito dalle diverse implementazioni dei diversi operatori che sono tra loro tutte equivalenti dal punto di vista logico/semantico (vuol dire che tutte le implementazioni diverse dello stesso operatore forniscono lo stesso risultato) MA non fisico (quindi anche l'elaborazione dei dati).

“Spazio di ricerca” nel senso che: quando ho un algoritmo di ottimizzazione, lo spazio di ricerca è l'insieme delle diverse alternative, tra cui mi muovo per cercare di minimizzare una funzione. Ho un insieme di cardinalità elevata di alternative (ad esempio nel caso di diversi operatori combinati) all'interno nel quale devo risolvere un problema di ottimizzazione, ovvero trovare la soluzione con costo minimo all'interno dello spazio delle soluzioni. Le diverse soluzioni implicano costi diversi.

Per raggiungere l'obiettivo di trovare la soluzione più economica: serve la possibilità di **enumerare** i diversi piani di valutazione della query (query plan), ossia le diverse procedure (sequenze di operazioni), ciascuna con una sua implementazione. Deve poter enumerare e quindi deve poter tradurre la rappresentazione dichiarativa in più equivalenti rappresentazioni procedurali della stessa query. Dopo di che ha bisogno di un algoritmo di ricerca sullo **spazio** (dato dall'insieme di piani alternativi all'interno del quale devo cercare) che cerchi però di evitare una scelta esaustiva proseguendo per "potature" e seguire euristiche per scegliere le parti da ignorare. Le parti da ignorare sono quelle che con probabilità altissima non sono promettenti.

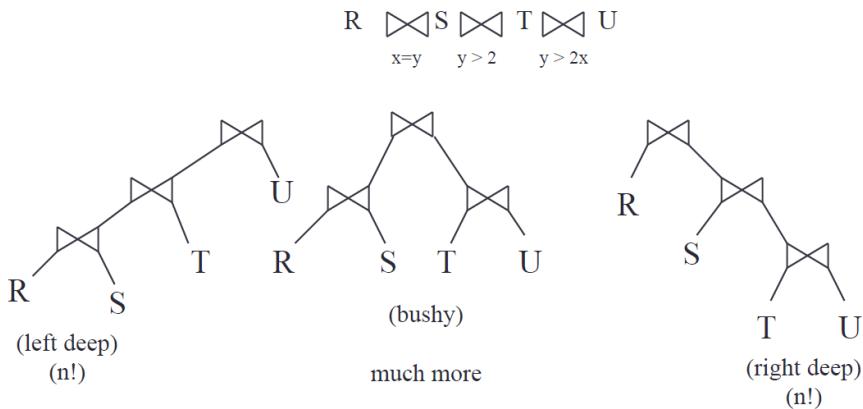
Per stimare il costo di una query, e per enumerare i piani relativi alla query stessa, ho bisogno di un modello rispetto a cui rappresentare il piano, ad esempio in **linguaggio naturale**, ma più efficacemente ho bisogno di un modello mediante il quale rappresentare le query. Il modello a cui si fa riferimento è un albero che prende il nome **Query Tree**.

Query tree: è una rappresentazione, sottoforma di un albero, del **work flow** (flusso di lavoro) della query. Il flusso di trasformazione che i risultati parziali subiscono per arrivare al risultato finale. Le foglie sono sempre gli operandi di base, quindi le relazioni secondo cui la query viene applicata, e i nodi interni sono gli operatori a cui corrispondono le diverse implementazioni. Possono essere relazioni o il risultato di altre scansioni. Si passa dalla versione dichiarativa (query, la versione che dichiara il flusso di dati esplicitando gli operatori da applicare) all'albero fisico di valutazione della query che dice quale implementazione viene scelta per realizzare le operazioni. Ciascun statement dell'algebra relazionale corrisponde a un query tree logico (l'algebra relazionale è già procedurale), quindi ciascuna procedura corrisponde ad un query tree logico e uno stesso query tree logico può corrispondere a diversi query tree fisici che corrispondono a diversi modi di implementare gli operatori citati nel query tree logico. Il goal è partire da uno statement SQL e arrivare a un query plan fisico che consenta di restituire i risultati dichiarati dallo statement.

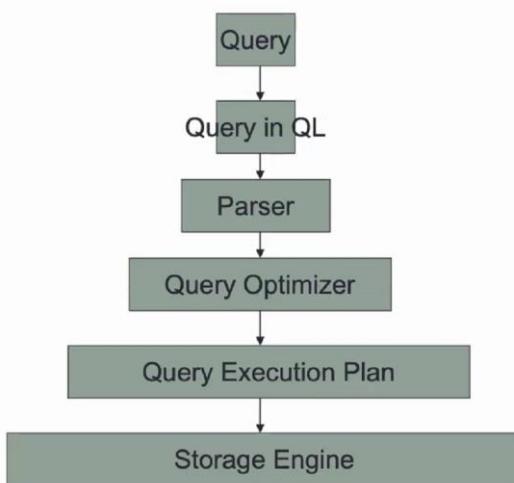


Se non si hanno informazioni specifiche per valutare la query, tipicamente ci si basa sull'aspetto dell'albero. In particolare, quando si valuta la combinazione di più join l'aspetto dell'albero è importante. Se uso strutture profonde a sinistra, qualora sulle relazioni coinvolte esistano degli indici, la struttura profonda a sinistra mi consente di usare più pesantemente gli indici. Inoltre, **left deep** mi consente di avere implementazioni non blocking perché tipicamente gli algoritmi di join sono fatti che per ogni record/pagina/blocco della relazione di sinistra, ciclo completamente quella di destra quindi per partire devo avere completamente quella di destra e con il **right deep** non è possibile avere l'intero argomento di destra fino a quando la query non venga interamente valutata.

Selects embedded in the Join



Query processing stages:



Query Optimization:

- Heuristic-based Query Optimization:
 - o perform cheaper operations first
 - o cheaper operations eliminate some of the inputs, hence more expensive operations need to deal with a smaller number of inputs.
 - Cost-based Query Optimization

È quell'operazione che ci porta a individuare, tra le possibili alternative per l'implementazione delle query, un'alternativa di relativamente basso costo (ossia idealmente meno costosa ma concretamente è un'alternativa di costo contenuto). Bisogna quindi passare da una query dichiarativa ad uno specifico query plan (piano di valutazione della query). Ci sono numerosi piani logici che possono corrispondere alla stessa query, dove i piani logici sono sequenze di operatori la cui valutazione porta al risultato della query a prescindere dalle implementazioni scelte. Ci sono numerosi piani fisici, ciascuno corrisponde a un piano logico ma la corrispondenza non è 1:1, ossia dato un piano logico possono corrispondere più piani fisici (quindi più rappresentazioni fisiche) perché alla stessa sequenza di operatori logici possono corrispondere diverse implementazioni e quindi diversi piani fisici. Tra tutti questi piani, l'obiettivo è trovare un piano ragionevolmente economico, per farlo bisogna avere un modello dei costi, un algoritmo di ricerca nello spazio

dei piani. Tipicamente, per fare una scelta, ci si basa su considerazioni di tipo euristico (in particolare è possibile anticipare il più possibile le operazioni che riducono la dimensione dell'output come proiezione e selezione piuttosto che join).

Nota:

- **Valutazione dei join:** ho un certo numero di strutture ad albero profonde a sinistra, dove questo numero è dato dal fattoriale del numero di relazioni coinvolti.

Ad esempio, se ho un join con 8 relazioni allora ho $8!$ Alberi profondi a sinistra distinti che corrispondono allo stesso join.

Dato che la quantità di scelte è alta, quello che in genere fanno gli ottimizzatori è limitare la propria attenzione a una specifica struttura, quindi non genera i **right deep** ma solo i **left deep**. I left deep lasciano maggior margine di ottimizzazione, nel senso che laddove le relazioni argomento del join prevedano l'esistenza di un indice sull'attributo del join, le posso sfruttare soltanto se queste relazioni di partenza sono operandi diretti del join e sono posizionati a destra (ovvero se ho un indice sulla relazione R lo posso sfruttare se sto facendo $S <> R$ ma NON se sto facendo $R <> S$ perché l'algoritmo di Nested Loop Join prevede che si possa utilizzare l'indice eventualmente esistente solo sulla relazione interna (quella di destra)).

Se ho un albero profondo a sinistra, tranne la prima foglia a sx del livello più profondo, tutte le foglie sono a destra dell'operando. Non avviene la stessa cosa nell'albero profondo a destra perché gli argomenti di destra sono risultati parziali di dati già fatti quindi l'indice si può sfruttare solo nella foglia più profonda a destra dell'albero.

Il right join è anche blocking per l'implementazione degli algoritmi, la relazione interna deve essere interamente disponibile durante il join bloccante e in questo caso a destra si hanno solo risultati parziali (non si può proseguire col join finché non è stato prodotto l'intero risultato di destra, non si possono usare gli indici avere valutazioni simultanee di operatori diversi che sfruttano i risultati parziali).

Heuristic VS Cost based operation: utilizzare queste euristiche, piuttosto che valutare tutte le soluzioni, che costo può avere? È chiaro che escludendo a priori certe strutture si introducono degli errori, quanto può essere grave e intollerabile introdurre degli errori? Potrebbe essere superfluo cercare la soluzione ottima quando alla fine ci si basa su un modello dei costi che è già lui stesso un'euristica ("grossolano"). Anche le ipotesi sulla distribuzione dei valori, su cui ci si basa per implementare la selettività, restituire i costi ecc., sono tutte approssimazioni/stime quindi ha senso fare dei calcoli esatti (e costosi) solo quando si ha la consapevolezza che siano precisi. Si tengono già le statistiche, tenere le statistiche sarebbe molto costose tenerle aggiornate. Se un'analisi precisa si basa su statistiche grossolane, allora l'analisi non è più precisa e non ha ragione di sceglierla.

Cost based query operation:

- Algebraic Transformations
 - commutative
 - distributive, etc.
- Operator Transformation
 - nested loop Vs sort merge join
- Operator insertion

- We are optimizing for cost of query processing
 - Cost is described in terms of disk access
 - Database keeps statistics about relations, tuples, page sizes.
 - Database also keeps index and sorting information
- Query optimizer uses “cost model” to guess query execution cost for a possible query execution plan
 - And chooses a plan which is least costly according to the cost model.
- Cost Estimation
- Plan enumeration/search space generation
- Search algorithm

Cost estimation: significa stimare la dimensione, ovvero il volume inteso come il numero di pagine che andrà a occupare del risultato dell’operazione.

- SCAN: what is the cost of reading one table from the disc?
- FILTER: what is the cost of doing a selection?
- JOIN: what is the cost of joining two tables?

Per ogni sotto-query devo stimare il costo, inteso come il numero di accessi a disco read/write che la sotto-query comporta e la dimensione del risultato prodotto.

Stima dei costi (locali): stima in termini di dimensione del risultato prodotto. È importante sapere la quantità di record in modo da poter stimare l’input dell’operazione successiva. Per fare una stima del costo ho bisogno di sapere quanto è grande il risultato, il suo input e quanto è selettivo l’operatore che si sta valutando.

- We need to estimate the number of disk I/Os.
 - Logical operators / physical operators
 - sizes of intermediate relations
 - ordering of operations (such as joins)
- How do we estimate cost of operations and sizes of resulting relations?
 - $B(R) \leftarrow$ number of blocks
 - $T(R) \leftarrow$ number of tuples
 - $V(R.a) \leftarrow$ number of distinct values for an attribute “a”
 - $V(R.[a_1-----a_n]) \leftarrow$ number of distinct values when $a_1-----a_n$ are considered together.

Nota: $V(R.a)$ corrisponde al numero di valori distinti dell’attributo “a” nella relazione “R”. $V(R.a) = T(R)$ se l’attributo “a” è una chiave.

- Cost Estimates
 - I/O cost + CPU cost
- Scan Cost
 - $B(R)$ (unsorted)
 - $B(R) + \# \text{of index pages}$ (sorted on a clustered index)
 - $T(R) + \# \text{of index pages}$ (sorted on a non-clustered index)
- (We already talked about the affects of database buffers)

Example use of statistics:

- Projection
 - easier
 - omit the attributes that are not included in the projection
 - recalculate how many blocks would be needed to store the resulting tuples.
 - What if distinct is specified?
- Selection
 - $S = \delta_{a=c}(R)$ when "c" is a constant
 - $T(S) = \frac{T(R)}{V(R.a)}$
 - $S = \delta_{a < c}(R)$ when "c" is a constant
 - $T(S) = \frac{T(R)}{V(R.a)} [V(R.a)-1]$ or $T(S) = T(R)$????
- $S = \delta_{c_1 \text{ or } c_2}(R)$
 - Assume c_1 and c_2 are independent
 - $T(S) = \frac{T(R) * [1 - (1 - \frac{n}{V(R)}) * (1 - \frac{m}{V(S)})]}{T(R) T(R)}$
 - where "n" is the number of tuples satisfying c_1 , and "m" is the number of tuples satisfying c_2
- Join
 - $R(x,y) \bowtie S(y,z)$
 - $T(R \bowtie S) = 0$
 - $T(R \bowtie S) = T(R)$
 - $T(R \bowtie S) = T(R)T(S)$
 - $T(R \bowtie S) = \frac{T(R)T(S)}{\max(V(R,y), V(S,y))}$
 - $R(x,y_1, y_2) \bowtie S(y_1, y_2, z)$
 - $T(R \bowtie S) = \frac{T(R)T(S)}{\max(V(R,y_1)V(S, y_1))\max(V(R, y_2)V(S, y_2))}$
- Alternative: Keep Histograms!
- Join-cost
 - Nested Loop join

$$\text{scan}(R_1) + T(R_1) * \underline{\text{scan}(R_2)}$$
 - Sort-merge join

$$\text{sorted-scan}(R_1) + \text{sorted-scan}(R_2)$$

Note sui costi:

- **Proiezione:** è semplice, devo leggere tutto il file e restituire un record per ciascun record presente nel file, perciò la stima dei record in output è immediata.
- **Selezione:** la stima della cardinalità dei risultati dipende dalla condizione di selezione, in particolare abbiamo condizioni di tipo diverso:
 - **attributo = costante.** Si ipotizza una distribuzione uniforme del numero di record per valore, quindi se avessi 1000 record e 100 valori distinti nella relazione per un certo attributo, il numero di valori pari all'attributo "a" sarebbe $\text{cardinalità_relazione}/\text{num_val_distinti_a}$. Questa stima potrebbe esser migliorata non basandosi su una distribuzione uniforme usando ad esempio degli histogrammi, è molto costoso però tenerli aggiornati quindi il gioco non vale la candela.
 - **attributo < costante.**
 - **attributo <> costante** (il simbolo è “**diverso**”). Se il numero di valori distinti è molto ampio e quindi ci sono pochi record per ogni valore, si può assumere che la cardinalità dell’output sia uguale a quello della relazione di partenza.

$$\frac{T(R)}{V(R.a)}$$

$$[V(R.a)-1]$$

Dove il -1 corrisponde alla costante che si vuole escludere, si assume che il valore distinto appaia in pochissimi record (uno solo), se ad esempio si ipotizza l’apparizione della costante in 1000 record allora sarà -1000.

- **c1 or c2 (disgiunzione).** Tipicamente si gestiscono i casi disgiuntivi con de-morgan. $n/T(R)$ è la percentuale di record della relazione che soddisfano la prima relazione. Se n è il numero di record che soddisfa la prima relazione, $1 - n/T(R)$ è la probabilità di record che NON soddisfano la prima condizione e la si può vedere come la probabilità di un record della relazione R di non soddisfare c1. Se m è il numero di record che soddisfano c2, $1 - m/T(R)$ è la probabilità per un record della relazione R di NON soddisfare c2. Vedendo le condizioni c2 e c2 come indipendenti (ipotesi), il prodotto della due probabilità di NON soddisfare c1 e NON soddisfare c2 dà la probabilità di NON soddisfare nessuna delle due (prodotto della congiunzione delle due parti). $1 - (\text{prodotto fatto prima})$ dà la possibilità che non sia vero che non soddisfo nessuna delle due e se non è vero questo, allora ne soddisfo almeno una delle due (eventualmente entrambe).
- **Join:** potrebbe capitare che l’output abbia la stessa cardinalità di una delle due relazioni, accade l’attributo di join è chiave. L’output ha cardinalità zero se ciascun record della prima corrisponde a ciascun record della seconda (accade quanto tutto corrisponde a tutto, ad esempio “diverso” da un valore che non esiste nella relazione). Il caso intermedio è quando il join ha una cardinalità pari a quello del cartesiano scalato rispetto al numero di valori distinti che l’attributo di join ha in uno delle due relazioni (il massimo tra quelli dell’uno e dell’altro perché i diversi valori distinti determinano dei “gruppetti” di cartesiano per cui non tutti fanno match con tutti ma ciascun record fa match con tutti quelli ha che hanno il suo stesso valore quindi “si formano tanti gruppetti distinti quanti sono il numero di valori di uno e dell’altro”.

Nota: dal punto di vista delle statistiche non si valutano le correlazioni tra le condizioni (valutando le condizioni come tipicamente indipendenti anche se non è detto che sia corretto in maniera effettiva). Ci

dobbiamo accontentare di “stime non pessime” perché tanto l’ottimo non sarebbe comunque preciso oltre che costoso.

Le assunzione alla base del modello dei costi: il costo relativo al risultato, in termini di cardinalità dei risultati generati, prescinde dalla specifica implementazione della query che è stato utilizzato e quindi è funzione solo dello specifico operatore e della specifica semantica che è stata implementata.

Assumptions of cost-based query optimization

- Independent of how a query is executed we get the same number of results:

$$\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$

Nota: cambiando l’implementazione si ha lo stesso output ma non è detto che il costo, in termini di pagine lette/elaborate, sia identico. Questo (riferito all’immagine appena sopra) perché è possibile che abbiano indici diversi, selettività diverse. È sempre bene anticipare il più possibile l’operatore che restituisce il risultato parziale con cardinalità più bassa rispetto alle alternative.

Principio di ottimalità delle sotto query: è il principio che regola il comportamento dell’ottimizzatore. È un principio alla base del comportamento degli ottimizzatori che per ottimizzare la query globale va a ottimizzare le sotto query, succede sempre così? No (non è sempre vero ma non è un’euristica).

Attenzione: non tiene conto che ci possa essere del vantaggio dato dalla conoscenza reciproca degli operatori.

Per ottimizzare a livello globale una query composta da un certo numero di sotto query, posso considerare individualmente il costo delle diverse sotto query e, dal momento che sommo i costi delle diverse sotto query, sceglierò come piano globale quello composto da sotto query che complessivamente hanno costo minimo. Scompongo il problema in sotto problemi.

Assumptions of cost-based query optimization

- Principle of sub-query optimality: An optimal plan for a query includes optimal subplans for the subqueries

$$\begin{aligned} \text{cost}^1(\sigma_{\theta_1}(\sigma^1_{\theta_2}(R))) &= \text{cost}(\sigma^1_{\theta_2}(R)) + f(\text{size}(\sigma^1_{\theta_2}(R)), \theta_1) \\ \text{cost}^2(\sigma_{\theta_1}(\sigma^2_{\theta_2}(R))) &= \text{cost}(\sigma^2_{\theta_2}(R)) + f(\text{size}(\sigma^2_{\theta_2}(R)), \theta_1) \end{aligned}$$

Questa formula dice che il costo del risultato dell’applicazione della selezione rispetto a θ_1 al risultato dell’applicazione della selezione rispetto a θ_2 di R , lo ottengo calcolando il costo della selezione interna

interna e sommando il costo della selezione esterna ma il costo della selezione esterna è funzione della condizione di selezione teta1 e della dimensione del risultato di quella interna.

Quando bisogna scegliere, si sceglie la più economica a livello locale. Un piano ottimo è quello che ha i sotto piani ottimi (si può usare la ricorsione):

Assumptions of cost-based query optimization

- Principle of sub-query optimality: An optimal plan for a query includes optimal subplans for the subqueries

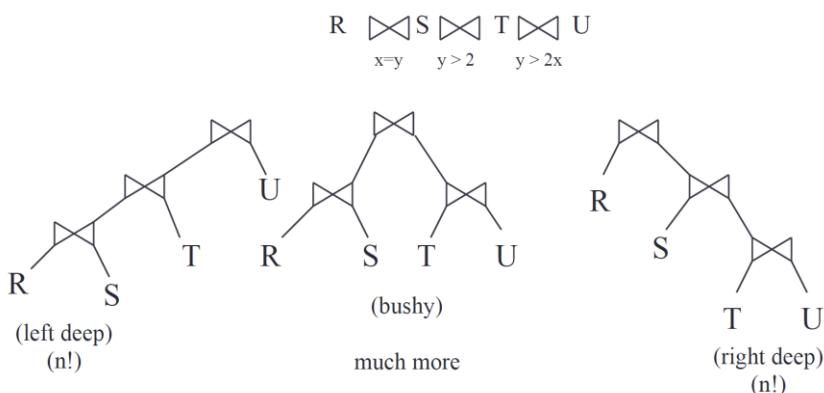
$$\cos t^1(\sigma_{\theta_1}(\sigma^1 \circ \sigma^2(R))) = \cos t(\sigma^1 \circ \sigma^2(R)) + f(\text{size}(\sigma^1 \circ \sigma^2(R)), \theta_1)$$
$$\cos t^2(\sigma_{\theta_1}(\sigma^2 \circ \sigma^1(R))) = \cos t(\sigma^2 \circ \sigma^1(R)) + f(\text{size}(\sigma^2 \circ \sigma^1(R)), \theta_1)$$

Example:

$$R_1(a,b,c) \bowtie R_2(a,d) \bowtie R_3(d,e) \bowtie R_4(c,e)$$

- Question: Which order of join is the best?
 - Important:
 - The join ordering problem is NP-Hard!
 - I.e., there is no known polynomial time algorithm
 - There is strong reasons to believe that there is no polynomial time algorithm

Selects embedded in the Join



Si sceglie left deep.

Recursion

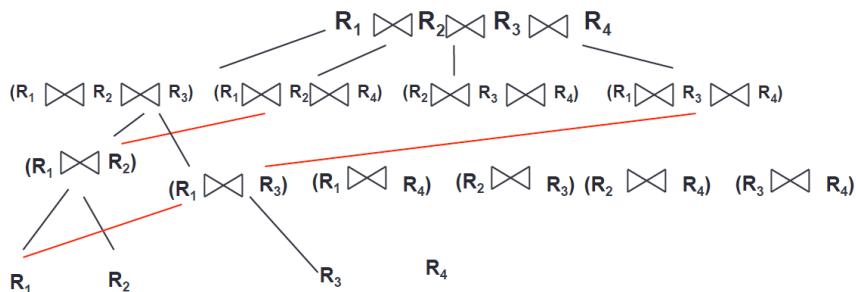
- Given $Q = R_1 \times R_2 \times R_3 \times \dots \times R_n$
 - for all R_i ,
 - find the cheapest plan for $Q_{(-R_i)}$
 - compute cost $_i = \text{cost}(Q_{(-R_i)}) \times R_i$
 - Pick the plan with the smallest cost
- There is a dynamic programming based algorithm that uses this recursion.

Recursion!!!

Programmazione dinamica: considero tutte le possibili implementazioni valutando quella più economica partendo dalle componenti più economiche. Costruisco il piano di valutazione di una sotto query di tre elementi ideale per ciascuna tripletta di relazioni. È una strategia che porta a creare una combinazione una volta sola.

Nota: con questa strategia non viene comunque risolto il problema del costo esponenziale associato al problema dell'utilizzazione. Non lo risolve perché è un problema esponenziale di sua natura, è esponenziale nel numero di relazioni, il numero di combinazioni diverse da gestire/valutare su cui bisogna calcolare i costi. Non ho risolto il problema del costo esponenziale ma dato il costo esponenziale, evito di aggravarlo ulteriormente, per via della gestione della ricorsione, gli stessi calcoli.

Dynamic Programming



Il problema dell'ottimizzazione delle query è ancora sotto studio data la complessità del tema.

Transaction Management Overview

Nota: guardare le slides.

Transazione: è una sequenza di operazioni su un db che vedo come "unità atomica". Sono operazioni per cui o tutte vengono concluse e rese persistenti su disco oppure tutte quante vengono annullate e si procede come se nulla fosse. È una sequenza di operazioni che NON può essere resa effettiva soltanto in parte.

Una transazione inizia con BEGIN e finisce con END.

L'operazione COMMIT è l'operazione che viene effettuata quando c'è certezza che i risultati accumulati (le modifiche effettuate dalla transazione che è arrivata alla fine), devono essere resi persistenti.

Una transazione può terminare in due modi:

- **COMMIT:** è andato tutto bene e i dati vengono resi persistenti su disco (non si opera su disco per ogni modifica fatta, troppi I/O quindi solo quando è necessario e alcune volte nemmeno quando si fa commit e di tanto in tanto si "scarica su disco").
- **ABORT:** termina la transazione e annulla le operazioni fatte fino a quel punto. Può essere dovuto da
 - o un'iniziativa dell'utente che blocca una transazione oppure
 - o può essere attivato da una decisione del sistema che può essere legata
 - a un guasto,
 - a una gestione di un'inconsistenza che si è potenzialmente creata (ad esempio al seguito di un guasto, era già stato fatto un pezzo ma la transazione non si è conclusa magari perché è mancata la corrente),
 - a situazioni di deadlock ...

Per quanto concerne le operazioni sul db, le operazioni su cui ci concentriamo sono le letture e le scritture. Le operazioni effettuate sui dati non sono rilevanti dal punto di vista del transaction manager, le operazioni che sono potenzialmente fonte di problemi (per quanto concerne l'isolamento e l'atomicità delle transazioni) sono quelle di lettura e scrittura su disco.

Note:

- Verrà utilizzato il termine "oggetto" per indicare in maniera generica i termini campo, record, pagina, tabella, database.
- Si parla di transazioni che, concomitamente, accedono a oggetti di granularità diversa.

Stato di un database: è l'insieme dei fatti che sono memorizzati nel database in un certo istante. Lo stato evolve per effetto delle operazioni di scrittura, mentre non subisce alcuna variazione per effetto delle operazioni di lettura. Quindi, le scritture determinano cambiamenti di stato del database e dunque posso vedere come una catena di dati che si susseguono nel tempo quello che è il **ciclo di vita del database**. In un certo stato del db, possono presentarsi senza problema diverse operazioni di lettura (ciascuna legge lo stato). Si vuole che ad ogni lettura vengano rispettati i vincoli di verità dei dati dello stato del db (quindi dei dati su db), ovvero che siano consistenti (vincoli di integrità). Ciascuna scrittura deve preservare la consistenza, ovvero deve restituire un nuovo stato del db a sua volta ancora consistente. Se c'è un abort, bisogna garantire che tutti gli stati temporanei, che hanno comunque garantito la consistenza ma senza arrivare alla fine della transazione, vengano completamente disfatti con l'operazione di **ROLLBACK** per riportare il db sullo stato consistente su cui la transazione aveva provato ad operare (attività di **Recovery** a fronte di **abort**). Si possono creare situazioni che vanno a violare il criterio di **isolation** delle proprietà ACID perché potrebbe capitare che si creino delle interferenze. Non tutti i casi di condivisione creano dei problemi

ACID: Atomicity, Consistency, Isolation, Durability.

Tipi di anomalie che possono presentarsi:

- **WR Conflicts:** reading uncommitted data. È un'anomalia che si presenta quando una transazione scrive un oggetto che viene successivamente letto da un altro. Si parla di **dirty reads**. Il problema è che la transazione T2 ha letto un dato che T1 ha scritto e va bene, però T1 fa abort e quindi il dato

scritto da T1 non è reso persistente quindi T2 ha fatto una lettura "sporca" del dato che in realtà non è mai stato ufficializzato.

- **RW Conflicts:** unrepeatable reads. T1 legge A, T2 scrive A poi T1 rilegge A e lo trova modificato senza che l'abbia fatto lui. T1 legge un dato che nel frattempo viene modificato da T2. È un comportamento indesiderato per il fatto che tra la prima e la seconda lettura di T1, si è intrufolato T2 che ha modificato i dati.
- **WW Conflicts:** overwriting uncommitted data. Una relazione scrive un oggetto ed è immediatamente sovrascritta da un'altra. Le modifiche apportate dalla prima transazione vengono perse. La prima write viene persa completamente perché viene sovrascritta.

Anomalies with Interleaved Execution

❖ Reading Uncommitted Data (WR Conflicts, "dirty reads"):

T1: R(A), W(A),	R(B), W(B), Abort
T2: R(A), W(A), C	

❖ Unrepeatable Reads (RW Conflicts):

T1: R(A),	R(A), W(A), C
T2: R(A), W(A), C	

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke 13

Anomalies (Continued)

❖ Overwriting Uncommitted Data (WW Conflicts):

T1: W(A),	W(B), C
T2: W(A), W(B), C	

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke 14

Scheduling Transactions:

- **Serial schedule:** schedule che non effettua l'interfogliamento tra le azioni delle relazioni. L'obiettivo è quello di eseguire le transazioni senza concorrenza. Mantiene l'ipotesi che se una tesi parta da un DB consistente, lo rilascia sempre consistente. Uno schedule seriale è chiaramente serializzabile perché è uguale a se stesso. Non ricorriamo sempre a esecuzioni seriali (che sono quelle che ci danno

massima garanzia) perché stiamo cercando di avere esecuzioni concorrenti. La serializzabilità è la base che viene considerata per il resto.

- **Equivalent schedule:** due esecuzioni delle transazioni che dato lo stesso stato del database in input, restituiscono lo stesso stato in output.
- **Serializable schedule:** concorrente serializzabile, è un'esecuzione che ha dell'interleaving (interfogliamento, quindi non seriale), ha un interfogliamento delle operazioni nel tempo ed è equivalente a una qualche esecuzione seriale della stessa. Equivalente vuol dire che dal punto di vista della trasformazione di stato che l'una comporta, la trasformazione de db fatta dall'altra è la stessa.

L'obiettivo è garantire la concorrenza (esecuzioni simultanee) ma al tempo stesso garantire la serializzabilità delle transazioni. La serializzabilità è importante perché una volta che so che uno schedule seriale ricevendo un db consistente lo lascia in una situazione consistente, se ho un'esecuzione concorrente che opera allo stesso modo e quindi lascia il db in uno stato in cui l'avrebbe lasciato un'esecuzione seriale, allora il db rimarrà consistente anche in questo caso.

In genere riconoscere schedule serializzabili è complicato allora siamo punti a capo. Per consentire un accesso concorrente che consente di avere un throughput più elevato, si deve dedicare tanto tempo per capire se lo schedule scelto è serializzabile. Il beneficio che si ottiene dalla concorrenza viene parzialmente consumato dal tempo dedicato all'analisi della serializzabilità. Tipicamente non si cerca uno schedule serializzabile ma ci si accontenta di un sotto insieme (una famiglia) di schedule serializzabili che sono schedule serializzabili dal punto di vista dei conflitti. Preso atto che ci possono essere situazioni di conflitto che si presentano quando c'è accesso concorrente ad uno stesso oggetto da parte di due transazioni, almeno una delle quali scrive, vengono gestite. La serializzabilità, dal punto di vista dei conflitti, prevede che nelle transazioni i conflitti che si generano nello schedule concorrente appaiano nello stesso ordine in cui sarebbero apparsi in uno schedule seriale.

Note:

- **Conflict serializable:** le due operazioni in conflitto vengono comunque eseguite temporalmente nello stesso ordine rispetto a come le avrei eseguite in uno schedule seriale. Ad esempio, T1 scrive A e T2 legge A.
Se uno schema è conflict serializable, è anche serializable (ovvero il primo è un sotto insieme del secondo). Hanno conflitti ma sono equivalenti a un'esecuzione seriale delle transazioni coinvolte. Nell'accettare il conflict serializability non rinunciamo alla serializzabilità, sono serializzabili per via del fatto che i conflitti siano esattamente in quell'ordine.
- **Conflict equivalent:** coinvolge le stesse azioni delle stesse operazioni
- **Serializzabilità:** dire che uno schedule concorrente è **serializzabile** vuol dire che lascia il database nello stesso stato in cui me l'avrebbe lasciato un'esecuzione seriale delle transazioni che invece hanno agito in maniera concorrente.

Conflict Serializable Schedules



- ❖ Two schedules are **conflict equivalent** if:
 - Involve the same actions of the same transactions
 - Every pair of conflicting actions is ordered the same way
- ❖ Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

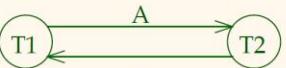
18

Example



- ❖ A schedule that is not conflict serializable:

T1: R(A), W(A)	R(B), W(B)
T2: R(A), W(A), R(B), W(B)	

 *Dependency graph*

- ❖ The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

22

Nota: è no conflict serializable perché ho dei conflitti che avvengono in direzioni diverse. In maniera seriale potrei fare T1 prima di T2 o il contrario, in questo caso T2 legge A dopo la scrittura di T1 sul dato ma T1 legge B dopo la scrittura fatta da T2.

Grafo delle dipendenze: dice in quale misura una transazione, rispetto a un certo oggetto, opera successivamente ad un'altra. Ad esempio (immagine precedente), T1 lavora prima di T2 su A (quindi comporrebbe un rapporto di precedenza) per soddisfare un certo vincolo. Implica che, per quanto riguarda A, la serializzabilità coinvolte T1 prima di T2.

Teorema: se uno schedule è serializzabile dal punto di vista dei conflitti, allora il suo grafo delle dipendenze è aciclico e viceversa.

Nota: quando il grafo delle dipendenze è ciclico, lo schedule non è serializzabile dal punto di vista dei conflitti.

Tecniche che consentono di riconoscere i casi in cui avremmo delle dipendenze cicliche per impedirle:

- **Locking:** pessimistico, è una delle tecniche che fanno prevenzione e non consentono che si presentino dei conflitti. È più conservativo, non consente di eseguire operazioni concorrenti se la

valutazione della concorrenza può dar luogo a un grafo delle dipendenze ciclico, questo vuol dire che quando può esser i un accesso concorrente a un oggetto si ritarda una delle operazioni per portare la situazione a essere serializzabile. È il modo utilizzato per tardare le operazioni che possono violare la ciclicità del grafo delle dipendenze. Chi vuole operare su un certo oggetto acquisisce il lock su quell'oggetto, quindi acquisisce un accesso/utilizzo esclusivo di quell'oggetto stesso, e soltanto al termine delle operazioni rilascerà il lock e consentirà agli altri di operarci, in questo modo si forza l'aciclicità del grafo e si forza la serializzabilità del grafo dal punto di vista dei conflitti dello schedule che si sta considerando. Si impedisce che il problema si ponga. Comporta tutta la gestione dell'eventuale deadlock e quindi bisogna discutere degli algoritmi che possano "rompere" l'attesa ciclica.

- **Timestamp:** ottimistico. Si basa sull'uso dei timestamp associati alle operazioni. È ottimistico perché si consente comunque agli schedule di eseguire e poi, prima di finalizzare il risultato delle transazioni che hanno agito concorrentemente sugli stessi oggetti, si verifica che lo schedule effettivamente generato è "**potenzialmente problematico**" (ossia se presenta conflitti) oppure no. Anziché bloccare le operazioni a prescindere (che implicherebbero del tempo), lasciano andare avanti e poi fanno il check alla fine. Si espone al rischio che il problema di ponga ma nel caso compaia non lo lascia propagare.

Strict 2PL (Strict Two-Phase-Locking): usa due tipi di lock. Quando bisogna operare su un oggetto bisogna acquisire un lock su quell'oggetto, quindi si dichiara l'intenzione di operarci. In Strict 2PL, al termine verranno rilasciati tutti i lock sugli oggetti, per questo è stretto. Il fatto che sia "stretto" garantisce che al momento che inizia a operare su un oggetto, nessun altro lo potrà fare fino a quando io non avrò fatto commit oppure abort (quindi uscito dall'insieme delle transazioni attive al momento).

- **LS (Lock Shared):** lock condiviso. Lo si cerca di acquisire quando l'operazione che si vuole fare sul dato non comporta inconsistenze sullo stato del db. Accade mediamente con le letture. Posso ottenere un LS solo quando sull'oggetto non ci sono ancora dei lock oppure ci sono ma sono tutti condivisi. Non c'è limite al numero di transazioni che possono leggere un certo oggetto, questo perché la lettura non implica una modifica dell'oggetto.
- **LX (Lock Exclusive):** lock non condiviso. Lo si richiede quando si vuole apportare una modifica all'oggetto. Esclusivo vuol dire "incompatibile con qualunque altro lock", che sia LS o LX non cambia. LX viene trattenuto dalla transazione per tutta la durata della transazione stessa.

Nota: l'unico caso di coesistenza di lock sullo stesso oggetto è solo tra LS.

Presenza: 24/05/21

Domande fatte a lezione:

- System catalog: è un file, una componente del dbms che tiene i metà sulla situazione dei file (oltre le statistiche degli accessi, ci sono le informazioni metà sulla struttura dei file). È quella parte dello schema che tiene ad esempio le informazioni sullo schema (come sono gli attributi, com'è indicizzato, lunghezza fissa/variabile, ecc.). Serve ad esempio quando bisogna effettuare una valutazione. Ex: a fronte di 500 record quanti valori diversi assume, serve per stimare (ad esempio) mediante euristiche le cardinalità dei join.
- Conservative 2PL: è 2PL non strict, quando si comincia a rilasciare i lock non li si possono più acquisire.

Presenza: 28/05/2021

Domande fatte a lezione:

Esame: possibili domande citate a lezione

- Domande alla fine dei capitoli
- Modello gerarchico e reticolare: sono troppo vincolati dai puntatori che rendono gli algoritmi poco efficienti. Il puntatore è di per sé inefficiente perché rendono gli accessi **IMPREVEDIBILI**, ovvero si sa che un puntatore punta ma non si sa dove va fintanto che non lo si ha analizzato e si ha raggiunto il valore. Queste imprevedibilità (dal punto di vista sulla gestione del db) hanno effetti pesanti sull'ottimizzatore che riesce ad ottimizzazione che query e l'organizzazione per il fatto che c'è una certa regolarità dei dati (ma così non avviene se ci troviamo in situazioni differenti).
- Si può dire che i db ad oggetti sono db NoSQL? Non proprio, OSL non è NoSQL ad esempio. I NoSQL sono piuttosto dei linguaggi indicati per i big data (grosse moli di dati, eterogenei, senza schemi), reti sociali ecc., tipicamente vanno bene per i dati di natura gerarchica. I NoSQL sono sistemi in cui tipicamente ci si accontenta di 2 su 4 requisiti ACID, con gli oggetti ci sono ancora.

- Indicare almeno tre casi di inter-dipendenze tra i moduli del DBMS discutendo le ragione per cui i moduli citati non potrebbero essere definiti in isolamento l'uno dall'altro
- Query evaluation: tipi di indici (gerarchico, hash, cluster, un-cluster, ecc.) utili ai vari operatori
 - Perché per alcuni tipi di dati non sono adatti database relazionali?
 - È possibile definire un hash-index come clustered?
 - Supponendo di avere un indice di tipo B+tree su dei dati, e supponendo che questo indice non renda efficienti le query, quali potrebbero essere le cause? Come risolverle? (almeno due)
- Descrivere quali pattern d'accesso vengono utilizzati nell'algoritmo sort-merge-join, e come vengono allocati i frame del buffer secondo l'algoritmo DB-min in questo caso.
- Descrivere almeno due casi in cui le scelte alla base del funzionamento di un modulo (buffer manager, disk manager, ecc) ha effetti sensibili su altri moduli a lui collegati.
- Quali sarebbero i problemi causati dall'inversione della fase di redo e della fase di undo nell'algoritmo di crash recovery?
- Quali sono alcune tecniche per l'operazione di eliminazione dei duplicati in una relazione?
- Quali sono le differenze nei costi di LRU e dell'algoritmo del working set in merito alla gestione della coda delle frequenze?
- Per quale motivo è opportuno salvare su disco il log invece che le pagine modificate? Evito un grande numero di accessi casuali dato che il file di log è append-only. Inoltre, posso scrivere su disco in modo "compatto" per cui è la strategia più efficiente. La scrittura di N modifiche sul log, in genere, coinvolge la scrittura di una sola o poche pagine, mentre scrivere N modifiche potenzialmente coinvolge N pagine diverse
- Elencare 2/3 esempi di interazione tra i moduli del buffer manager, ad esempio il checkpointing
- Differenza tra abort e crash? Hanno una differente implicazione e gestione.
- Serverebbe mantenere all'interno del log le informazioni relative alle query fatte? Se sono interrogazioni con sola lettura no perché a sola lettura non cambia lo stato del DB.

- Come funziona il block nested?
- Se ho N blocchi, quante pagine bisogna leggere? È richiesto quindi istanziare la formula sul caso specifico, è importante motivare la risposta.

*ESERCIZIO: data la sequenza,
descrivere il comportamento delle 3
opzioni indicate in seguito:*



- ❖ Sequence S1: T1:R(X), T2:W(X), T2:W(Y),
T3:W(Y), T1:W(Y), T1:Commit, T2:Commit,
T3:Commit

option1. Strict 2PL with timestamps used for deadlock prevention.

option2. Strict 2PL with deadlock detection. (Show the waits-for
graph in case of deadlock.)

option3. Conservative (and Strict, i.e., with locks held until end-of-
transaction) 2PL.

Prof.ssa Rosa Meo (3cfu)

Introduzione ai NoSQL Databases

Per un lungo periodo sono stati utilizzati per sfruttare le loro proprietà di persistenza dello storage nelle applicazioni SW. Ci danno la garanzia di memorizzare in maniera persistente i dati (che sono di grande valore). I DB sono una sorgente persistente e concedono più funzionalità dei FS. I database sono un'alternativa più robusta e flessibile che i File System, questo perché sono stati concepiti per gestire un numero maggiore di dati.

I DB permettono l'integrazione di molteplici applicazioni sw che usano il DB come strumento di accesso ai dati unico alle applicazioni. È possibile quindi accedere ed effettuare operazioni sui dati che risulteranno persistenti anche nelle altre applicazioni.

I DB permettono l'utilizzo delle transazioni (**proprietà ACID**) e il **roll back** in caso di errori. Le transazioni sono arrivate per affrontare il problema dell'accesso concorrente alla memoria.

Transazione: ambiente naturale in cui si possono effettuare le operazioni

Proprietà ACID: sono utili quando si usano applicazioni in cui bisogna accedere in maniera concorrente ai dati, sia che il set di dati sia distribuito su più nodi che centralizzato sullo stesso server.

- Atomico
- Consistente
- Isolabile
- Durabile

Roll back: possibilità di disfare una transazione e tornare allo stato precedente alla sua esecuzione.

DBMS: è il garante della consistenza e dell'integrità dei dati, a lui viene riversata la responsabilità di gestir i dati e quindi effettuare i controlli su di essi per garantire le proprietà ACID (oltre alle altre operazioni che esegue). Il motivo principale per cui vengono utilizzati i DBMS è perché è una garanzia si integrità e consistenza dei dati e viene usata come sorgente di condivisione e integrazione.

Nota: i DB relazionali sono stati di grande rilevanza negli ultimi 20 anni perché permettono **OLTP** (Online Transaction Processing) e hanno un throughput di transazioni al secondo molto elevato, ma anche perché hanno l'SQL che è uno standard per interagire con i dati, quindi le applicazioni sono più "portabili" in quanto ogni SW che sia in grado di interfacciarsi mediante SQL può usarli (indipendentemente da come è stata scritta l'applicazione)

Integrazione di DB condivisi: il DBMS gestire i dati e le applicazioni interagiscono con il DBMS.

Difetto dell'uso dei DB come strumento di integrazione: il DB risulta essere una risorsa comunque e gli interessi di ciascun team potrebbero essere in conflitto con quello degli altri team e quindi lo schema dei DB inizia a essere inutilmente complesso. Essendo lo schema “condiviso”, i teams si devono coordinare e si potrebbero presentare una serie di aspetti negativi come la lentezza.

Difetto del DB (in particolare del relazionale):

- Accetta solo valori atomici, quindi è un modello un po' rigido perché accetta i dati solo del modello relazionale e per ottenere i dati in più tabelle si è costretti ad utilizzare delle join sugli attributi chiave, le join però risultano essere delle operazioni molto lunghe e complicate da fare.
- Si parla quindi di **Impedance mismatch**, cioè le applicazioni (quando lavorano) hanno a disposizione strutture dati molto più flessibili ed evolute (come la struttura dati ad oggetti oppure le strutture dati dinamiche come gli alberi e le liste) mentre il DB relazionale ha come unica struttura dati (e formato) la tabella. Pertanto, ci può essere la necessità continua di tradurre la rappresentazione della struttura dati del programma, nella rappresentazione dei dati accettata dal modello relazionale (continua traduzione del formato dei dati). Potrebbe essere molto più comodo avere un db che accetta direttamente le strutture dati usate dal programma come JSON e XML.

Application database: è una soluzione che consiste nell'avere un db in locale, quindi non come quello accennato precedentemente volto all'integrazione di applicazioni sw per la condivisione dei dati MA un db privato dell'applicazione sw che quindi è libero di mantenere il **data model** (il modello dei dati) più comodo al programmatore (senza la continua necessità di dover in continuazione tradurre dal formato dell'applicazione dati a quella del db). In questo caso la responsabilità dei dati è del team di sviluppo dell'applicazione quindi non si presenta la necessità di dover negoziare (in maniera continuativa) l'utilizzo del db. Essendo il DB utile direttamente all'applicazione, è possibile scegliere l'architettura più adatta all'applicazione stessa.

SOA (Service Oriented Applications): permette di utilizzare lo “strato web” come strato di presentazione dei dati. È l'interfaccia (utilizzando il protocollo HTTP) può essere usata anche come interfaccia che l'applicazione può esporre alle altre applicazioni. Quindi c'è la possibilità di utilizzare l'interfaccia come mezzo per raggiungere i dati senza doversi direttamente interfacciare al DB.

Clusters Support: molte applicazioni possono lavorare su un maggior numero di server che possono comunicare tra di loro e messi in un **cluster di macchine** che via rete possono essere collegate tra loro e partizionarsi la collezione dei dati che possono essere modificati dall'applicazioni sw. Il **supporto dei cluster** è molto comune quando l'applicazione deve gestire un enorme volume di dati e quindi non basta un singolo server per gestire tutto il volume dei dati (spesso chiamati **Big Data**). Questo problema è apparso con l'uso esponenziale del web che richiede aggiornamenti di infrastruttura (sw e hw). Usare una macchina super prestante è costoso, mentre usare un **cluster di servers** è più economico perché ogni macchina è un **commodity server**, cioè la potenza non dipende dalla macchina ma da quante macchine vengono introdotte nel sistema (si parla di migliaia di server in cluster tra loro). Vengono usati dalle grande aziende dell'era “**dot com**” per poter utilizzare in maniera distribuita e parallela i dati e reggere il peso delle richieste che arrivano al server. Il problema è che questa tecnologia fa fatica ad andare d'accordo con i modelli relazionali lavorando in maniera **distribuita** perché il concetto di **transazione** nel modello relazionale, è un concetto **forte**. La probabilità che almeno una macchina all'interno di cluster si rompa quotidianamente è quasi certa (si parla

di numeri alti di server), quindi non è fattibile l'idea di salvare i dati su una macchina sola. Quindi si replica lo stesso dato in maniera ridondante su più macchine (transazione distribuita), per scrivere il dato in maniera ridondante su più macchine bisogna garantire che la transazione abbia successo e completi la scrittura su tutti i nodi in cui il dato è replicato. **SE** un noto va in down, la transazione non riesce a completare il proprio lavoro perché la richiesta di consistenza forte richiede che tutti i nodi siano stati contattati e abbiano rilasciato un feedback positivo sull'operazione di scrittura. Quindi i DB relazionali fanno **mismatch** (non vanno d'accordo) con il **cluster support** qualora i dati venissero distribuiti su più macchine cosa è che però è una realtà nelle situazioni in cui si ha dei cluster. È quindi richiesto un modello di organizzazione dei dati diverso che **rilassi le richieste di transazione forte** dei db relazionali.

Big Data: si presentano in situazioni reali in cui i dati arrivano con un flusso continuo e quindi un singolo server potrebbe non bastare per gestire tutto quel volume dei dati.

Modelli di distribuzione dei dati e gestione della consistenza nei NoSQL DB

Sono delle soluzioni nate dell'era **dot com** volte a risolvere i problemi dei db relazionali nei modelli distribuiti. Amazon ha proposto come soluzione **Dynamo** e Google invece **Big Table**. Sono dei DB colonnari che hanno un principio di memorizzazione dei dati a colonna (invece che a riga) che risultano essere più efficienti nel momento in cui i dati sono sostituiti.

I NoSQL DB sono emersi come un "nuovo movimento" che nasceva principalmente nei progetti Open.

L'acronimo **NoSQL** non indica un rifiuto a SQL ma piuttosto la possibilità di interrogare il DB con un linguaggio che non è solo (**Not ONLY SQL**) l'SQL, successivamente c'è stato un rifiuto a SQL e le proposte a modelli di rappresentazione e gestione di dati alternativi. L'alternativa non si è solo fermata al modello del dato ma anche ai principi che vengono garantiti dal DB (**ACID**, quindi i principi alla base delle transazioni) che sono stati messi in discussione e rilassati.

Molti di questi DB non hanno uno schema fisso cosa che invece è richiesta nel modello relazionale (non è possibile inserire i dati prima di aver stabilito lo schema delle tabelle).

Note:

- progetti open
- permette la distribuzione di dati su cluster di server
- interessati a rilassare le proprietà di consistenza per la necessità di dover distribuire i dati su più server

Data Model: quando si parla di data model in genere ci si riferisce a due "cose" diverse

- se si è programmati e si sta sviluppando un'applicazione sw, si parla del modello dei dati dell'app sw e quindi dello schema dei dati (i modello che l'app usa per gestire i dati), invece
- quando si parla di tecnologia dei DBMS, per modello dei dati si intende il modello del dato del DBMS che il sistema usa per salvare dentro di se i dati (il modello che il db usa per memorizzare i dati, ex ER). Si dovrebbe chiamare **metamodel** perché è il modello su cui il programmatore andrà a realizzare il proprio modello dei dati.

Modello relazionale: basato su una forma tabellare dove ciascuna tabella è una relazione, usano le righe (**tuple**) per rappresentare oggetti di interesse. Per descrivere gli oggetti si usano gli attributi (le colonne), ogni

attributo ha un valore (eventualmente nullo) ma non può rappresentare oggetti annidati (a meno del caso dell'Object Relational). Se si ha la necessità di far riferimento ad oggetti esterni si usa il concetto di chiave esterna come colonna che corrisponde alla chiave primaria dell'altra tabella, lo strumento che si utilizza è il join.

Quattro categorie di modelli usati negli ecosistemi NoSQL: il NoSQL fa un passo avanti rispetto al problema della referenziazione ad oggetti esterni mediante chiave esterna e quindi l'utilizzo dello strumento join. Ogni soluzione NoSQL usa un modello diverso.

- **key-value:** basato su coppie chiave-valore dove si può percepire come la chiave in un attributo della riga e il valore come il valore di quell'attributo per quella riga
- **document:** rinuncia ad avere una struttura ma sposa l'idea di avere una struttura **semi-strutturata**, quindi scrive in un documento testuale la struttura dei dati che si sposa meglio con una struttura annidata tipica degli oggetti (usando ad esempio le graffe per rappresentare l'annidamento di un'oggetto dentro un altro e usando lo strumento principale di memorizzazione di attributi dentro un oggetto come list e array). Ad esempio JSON.
- **column-family:** db colonnari che vedono la colonna come strumento principale di memorizzazione piuttosto che la riga, è adatto a quei db che hanno dei valori sparsi dentro le tabelle
- **graph:** si sposa bene per molte app come mediche, app che vanno a memorizzare la stessa rete dei contatti social, ecc.

Aggregate data model (modello dei dati aggregato): non necessariamente indica che si usa una funzione aggregata (ex la funzione di somma, conteggio, ecc.), è una collezione di oggetti che noi sappiamo che tipicamente viene acceduta insieme dall'applicazione sw. Ad esempio, in un db transazionale, è la collezione di oggetti che vengono acceduti prima di fare commit.

Si può definire l'aggregato come l'unità dei dati che vengono manipolati contemporaneamente dall'applicazione sw, quindi che vengono richiesti e aggiornati. Tipicamente nelle applicazioni sw si aggiornano i dati con una transazione **atomica**.

Nel caso di DB distribuiti, la collezione dei dati potrebbe essere distribuita su più server del cluster, quindi questo aggregato è l'unità di replica che potrei voler fare qualora voglia replicare più volte nel caso in cui un server vada in down ma si vuole garantire il servizio (e quindi mantenere le applicazioni sw sempre "vive" che non si possono bloccare a fronte di un server guasto e possano contattare un altro server).

Modello aggregato nell'esempio dell'e-commerce: vedere slides. JSON è una rappresentazione semi-strutturale e flessibile. Ad esempio, "billingAddress" può essere modificato a seconda dell'utente. Mettere le info dei prodotti negli ordini permettono ad esempio le attività di analisi, la scelta di design di rappresentare in maniera fortemente ridondante le informazioni avviene al seguito dell'assunzione delle seguenti ipotesi:

- ipotesi di non dover aggiornare spesso i dati
- di dover fare query quasi solo in lettura e
- di favorire le operazioni più onerose (quelle che vanno a coinvolgere una maggior quantità di dati)

Nota: le scelte di progettazione vanno a influenzare l'organizzazione dei dati in termini di schema e memorizzazione. Se cambiano le operazioni sul db bisogna cambiare il design poiché erano state favorite le operazioni precedentemente rilevate. Le scelte dipendono dallo specifico contesto applicativo e da come

l'app accede ai dati (in quale maniera li raggruppa), questo procedimento favorisce il procedimento in lettura (non in scrittura) dei dati per fare le statistiche.

Aggregate-ignorant VS Aggregate-aware Models:

- **Aggregate-ignorant:** cioè ignora la maniera con la quale i join verranno fatti dall'applicazione. Lo sono il Modello Relazione e quello a Grafo. È talmente atomico nella propria rappresentazione atomica dei dati che è molto flessibile (aspetto positivo), rappresenta separatamente i dati e poi se servono insieme effettua il join. Cambiando i requisiti dell'app basta cambiare l'SQL. Favoriscono la flessibilità da parte delle applicazioni di cambiare i requisiti e quindi possono cambiare il modo con cui accedono ai dati in maniera molto flessibile MA a **run-time** possono essere più lente perché devono effettuare i join. Nelle data warehouse spesso vengono usate le liste materializzate come strumento intermedio per pre-computare i join e memorizzare i risultati dei join come delle tabelle già computate. I DB NoSQL usano la stessa terminologia di liste materializzate quando hanno bisogno di pre-computare delle informazioni e metterle a disposizione del compilatore con la stessa filosofia MA non usando il modello relazionale.
- **Aggregate-aware Models:** (come il modello a documento JSON) facilitano le query in lettura e in particolare le query distribuite che girano su server in cluster tra di loro e non devono effettuare delle operazioni troppo complicate per andare a reperire questi dati perché tutti gli elementi che compongono un aggregato sono fisicamente residenti sullo stesso server. Quindi per eseguire queste query in lettura si fa accesso a un numero limitato di server che contengono le informazioni che servono.

Transazioni:

- **Modello relazionale:** ci permette di manipolare qualunque combinazione di riga in una transazione, basta fare una (o più) query che effettua l'operazione join. Per chiudere l'operazione ha bisogno che tutte le righe siano a disposizione contemporaneamente, SE uno di questi server che contengono queste righe non è a disposizione, la transazione non può chiudere e rimane appesa e blocca l'app sw che rimane in attesa della risposta.
- **Aggregate-oriented DB:** non c'è alcuna nozione di transazione, i dati sono già a disposizione di questo aggregato e il dato viene recuperato in lettura. Non c'è alcuna garanzia da parte del DB per la transazione, non tutti i db NoSQL hanno il concetto di transazione (poiché è il primo concetto che è stato rilassato), l'unica garanzia è che quei dati vengano recuperati insieme perché si trovano dentro il modello aggregato. Qualora una query abbia bisogno di accedere a più informazioni che risiedono in più aggregati, quella query si trova in difficoltà perché non ha la garanzia transazionale di poter lavorare in maniera consistente (atomica) su tutti i "pezzi di dati" distribuiti fra gli aggregati.

Aggregate data store: "data store" è il nome che viene dato ai "DBMS NoSQL". Tipi di NoSQL Database (differiscono soprattutto nel poter indicizzare la parte "valore" del dato):

- **key-value:** gli store key-value hanno una capacità molto efficiente di andare a ricercare il valore che corrisponde a una chiave. Hanno delle enormi mappe che memorizzano i valori indicizzando le mappe per la chiave e il valore è l'unica "cosa" che riescono a recuperare attraverso la chiave. C'è un'unica modalità di indicizzazione, tramite la chiave della coppia key-value. Non possono indicizzare i valori (anche se complicati). I valori vengono memorizzati in maniera "opaca".

- **document:** sono più sofisticati e alcuni riescono a indicizzare anche parte del documento testuale. Ad esempio, Riak che usa Apache Solar che è una particolare modalità di ricerca di tipo testuale che permette di andare ad cercare nei documenti testuali dei pezzi di documento e quindi viene utilizzato nelle applicazioni di **Natural Language Processing** che hanno bisogno di calcolare le frequenze delle parole nei testi.
- **column-family databases:** i database colonna si chiamano “column-family” perché nella definizione dei dati, raggruppa le definizioni in famiglie. Ad esempio Big Table di Google. Il concetto di “colonna” è fuorviante perché la memorizzazione dei dati non avviene in formato tabellare ma piuttosto con mappe chiave-valore. Va bene per tutte quelle soluzioni in cui si hanno oggetti con tante colonne e altri con poche (con celle in comune o senza, quindi ci possono essere colonne specifiche dell’oggetto), se venisse memorizzato in una matrice si avrebbe una matrice sparsa.
Questa struttura è più simile a una struttura a doppia mappa annidata
 - La prima mappa (chiave-valore esterna) è una struttura dati che memorizza le righe, quindi la chiave è l’identificatore della riga. Nella mappa esterna, il valore associato all’indeterminato della riga è la mappa interna.
 - La mappa interna va a memorizzare tante coppie chiave-valore quanti sono i valori degli attributi per quei valori di quella riga. Va quindi bene per quelle rappresentazioni sparse dove si possono avere poche colonne o tantissime colonne in modo molto disomogeneo da riga a riga perché ogni riga ha la propria mappa che va a memorizzare le colonne presenti in quella riga., è molto dinamico perché volendo si può aggiungere un numero a piacere di nuove colonne a ciascuna delle righe.

Questa soluzione si adatta a tutte le soluzioni in cui si hanno una soluzione sparsa.

Tipi di righe (questa terminologia dal db colonna Cassandra):

- **Fat rows:** sono le righe che hanno tantissime colonne (molto dense di colonne)
- **Skinny rows:** hanno un numero di colonne molto “raro” (un numero di colonne limitato)

Nota: Cassandra è un database NoSQL di tipo colonna.

Graph databases: modello di database il cui modello dei dati è un grafo, sono ideali per catturare delle situazioni dove le relazioni che connettono gli oggetti sono **eterogenee (molto diverse tra di loro)**. Nascono dalla necessità di lavorare guidato da aggregate-oriented data models con connessioni semplici. Le strutture a grafo sono composti da **nodi (nodes)** connessi da **archi (edges)**. I nodi sono semplici (contengono solo i nomi), la struttura è costituita da una ricca interconnessione fra nodi.

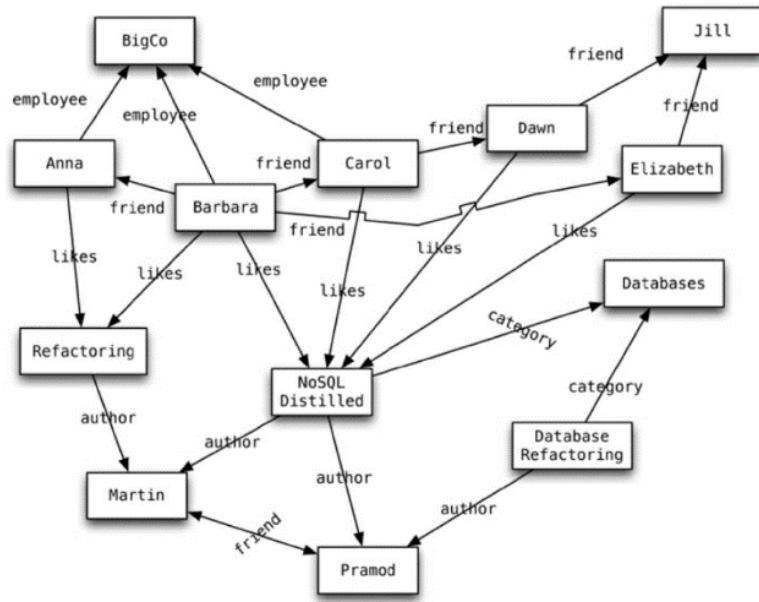
Graph databases are ideal for capturing any data consisting of complex relationships such as social networks, product preferences, or eligibility rules. In una social network viene costruita in maniera naturale una di queste reti.

Examples of graph databases:

- **FlockDB** is simply nodes and edges with no mechanism for additional attributes. Prevede nodi e archi semplicissimi che sono costituiti da nodi e archi con un semplice valore di “attributo” ma non prevedono meccanismi di assegnamento di attributi aggiuntivi né ai nodi né agli archi.
- **Neo4J** allows you to attach Java objects as properties to nodes and edges in a schemaless fashion. È uno dei più famosi db a grafo, permette di “attaccare” oggetti veri e propri in Java che poi si possono eseguire attraverso i metodi ed avete a disposizione i propri attributi come proprietà assegnabili sia

ai nodi che agli archi del grafo. Lo schema non è predefinito in anticipo, quindi si possono chiamare “senza schema” (schemaless). Si possono usare senza avere (in maniera preliminare) la chiara idea di come sono i nodi e gli archi.

- **Infinite Graph** stores your Java objects, which are subclasses of its builtin types, as nodes and edges. Memorizza oggetti java ma questi (quelli che permette di associare agli archi e ai nodi) sono delle sottoclassi dell'oggetto Java iniziale. Design OO.



Query:

- questo genere di db sono molto efficienti nel fare delle query su grafi molto grandi. In un db relazionale, le performance diminuiscono all'aumentare delle relazioni (join) mentre con i db a grafo le performance rimangono alte anche nei casi con numeri elevati di relazioni.
- i graph db sono molto “bravi” e veloci a fare interrogazioni complesse. Sono molto veloci ad attraversare un grafo in lettura perché sono stati progettati per favorire la lettura, non tanto la scrittura.
- le operazioni in scrittura (al tempo di modifica) sono un po' meno rilevanti per l'applicazione.
- db graph ha flessibilità nelle interrogazioni (che naturalmente essendo in lettura non effettuano modifiche dei dati)

Graph db e clusters: i grafi sono strutture molto connesse, quindi quando decidiamo di partizionare il grafo si favorisce una porzione del grafi a discapito di un'altra. Per questo motivo questo genere di db non è adatto a lavorare in cluster id macchine perché ciò richiederebbe di partizionare il grafo e questo non è così immediato farlo. Come di DB relazionali condividono la difficoltà di supportare transazioni ACID sulla struttura dati qualora questi dati venissero partizionati su più strutture del cluster (funzionano meglio lavorando in **single server**).

Vantaggi dello schemaless:

- With relational databases, we have to agree on the schema before we can store data

- With document stores, key-value stores and column-family stores, we can store in the value any data that we want associated to a key, or with any additional column.
- In graph databases, we have the freedom to add new edges, and new properties to nodes and edges.
- This turns out in more freedom in handling changes during development:
- as soon as we discover new features during the project we can add them
- we can stop storing some features as soon as we do not want them, without the risk of losing the data already stored

Svantaggi dello schemaless:

- The flexibility in schema turns out to an increased complexity in developing the software for parsing the objects content (it has to foresee the presence of each feature in any of its development stages)
- A schemaless database assumes an implicit schema (assumes that certain field names are present and have a certain meaning). This implicit schema is a set of assumptions about the data's structure in the code that manipulates the data.
- A schemaless database shifts the schema into the application code.
- Viene spostata la complessità dal modello al sw, un db senza schema in realtà sposta lo schema sul codice (è stato solo spostato dove la complessità risiede).

Soluzioni al fronte del problema di “parsificare” i dati che non hanno uno schema:

- si scrive un'applicazione sw che concentra in uno strato tutta la conoscenza dei dati, ovvero tutte le interazioni col db (incapsulamento di tutte le iterazioni col db).
- The other solution is to clearly delineate different areas of an aggregate for access by different applications. These areas could be the different sections in a document database or different column families in a column-family database. Sfruttare la struttura dati per andare a isolare certe porzioni dello schema. E se si utilizza un db orientato alle colonne, si possono sfruttare le colonne e la “storia” dello schema.

In ogni caso, cambiare i confini dei modelli basati su “aggregato” è una cosa complessa, così com’è complesso organizzare l’organizzazione “a colonne” di un modello relazionale perché è come gestire la migrazione di uno schema.

Viste materializzate: sono percepite dagli utenti finali come delle tabelle di base, in realtà sono interrogazioni eseguite real-time. Se (nell’ottica di favore le query in lettura) si vuole alleggerire il peso computazione di queste liste, si può gestire le liste materializzandole (salvando il risultato delle viste in maniera persistente oppure in cache) in modo che tutte le app che hanno bisogno di utilizzare questi dati possano non eseguire di nuovo le query. Nonostante di NoSQL db non hanno le viste, hanno delle query pre-computate o messe in cache.

Problema: come mantenere il contenuto delle viste materializzate aggiornate rispetto alle modifiche del db (bisogna mantenere allineati i dati)

Strategie per mantenere le viste materializzate:

- Approccio **eager**: viene aggiornata la lista materializzata quando viene effettuata una modifica dei dati che la riguardano
- Approccio **lazy**: verranno eseguiti dei **batch jobs** per aggiornare la lista materializzata (quindi non viene aggiornata ad ogni update)

Nota: la scelta dipende dalle necessità di chi commissiona e dalle necessità (ex quanto e quante volte bisogna accedere ai dati, con quanta precisione si richiedono i dati, quanto la query rallenta l'applicazione, ecc.)

Quindi la scelta delle viste materializzate subentra qualora il modello aggregato non sia più al 100% adatto nel rispondere a tutte le interrogazioni.

Real-time analytics: vengono effettuate su enormi quantità di dati, la cosa ideale è effettuare le operazioni in real-time senza avere il bisogno di doverle schedulare come quelle tipi batch (una volta ogni tanto). Le query real-time danno risultati più accurati in quanto vengono eseguiti sul momento e quindi fanno riferimento a dati "freschi". La **denormalizzazione** dei dati permette di velocizzare l'accesso ai dati di interesse.

Nota:

- con i db di tipo relazionale si parla di tabella di data warehouse in cui l'obiettivo della data warehouse stessa è di mettere a disposizione i dati in una forma tale da favorire queste query di BI (Business Intelligence). Nelle data warehouse i dati sono già memorizzati in maniera denormalizzata (in particolare nella tabella centrale che è la così detta "tabella dei fatti" dove avvengono le statistiche (analytics)).
- al di fuori dei db relazionali e delle data warehouse, noi potremmo realizzare i documenti store che forniscono degli indici che permettono di parsificare la parte **valore** associata al documento, e quindi si riesce a fare delle vere e proprie ricerche fini dentro al valore.

Data distribution:

- **Replication:** tiene molte copie dello stesso dato (replicarlo). NON ha senso replicarlo sullo stesso disco perché lo scopo è rendere l'applicazione robusta a fronte di guasti, se entrambe le repliche sono sullo stesso hw e l'hw si rompe -> entrambe le repliche non sono più disponibili. Obiettivo: tenere più copie dello stesso dato ma su più nodi.
- **Sharding:** partizionamento del dataset, vuol dire mettere dati diversi su nodi diversi. Questo serve per scalare l'applicazione sul volume dei dati (un singolo hdd non basta per salvare tutti i dati, ad esempio dei clienti, quindi si fa lo sharding). Ad esempio, il partizionamento dei clienti per area geografica (nord, centro sud), in questo modo non si hanno tutte le richieste centralizzate sullo stesso server (riduce quindi il traffico di rete).

Combinazione di tecniche di distribuzione:

- **first single-server**
- **master-slave replication:** gerarchie di server.
- **sharding:**
- **peer-to-peer replication:** i server sono tutti paritari

Single server distribution: caso più semplice a singolo server, corrisponde al "non distribuire" perché si ha un unico server. È il più semplice dal punto di vista della progettazione ed è quella raccomandata per la sua semplicità. Accetta ogni tipo di interrogazione in lettura e scrittura e va bene per qualunque tipo di db (sia quelli a grafo che sono difficili da distribuire che per gli altri come key-value e document).

Multiple servers sharding: partizionamento con lo sharding dei dati su più server (utilizzo di molteplici server ma con sharding). Abbiamo molti server e ciascun dato è allocato in uno solo di questi server (non ci sono più copie dello stesso dato), come avere un singolo server solo che vengono distribuite le interrogazioni geograficamente. Bisogna prevedere una modalità di distribuzione che permette di bilanciare il carico di interrogazioni che arrivano a ciascun server. Modello di tipo aggregato che ha l'obiettivo di progettare l'insieme dei dati che verranno memorizzati negli stessi data store e che verranno richiesti in lettura dalle interrogazioni evitando che la stessa interrogazione vada a interrogare diversi data server. L'**auto-sharding** potrebbe comportare alcuni problemi dovuti da

Load balancing: riguarda il come distribuire il carico in maniera bilanciata rispetto ai diversi server, bisogna cercare di distribuire in maniera il più possibile uniforme gli aggregati (che sono l'unità di lettura e di modifica dei dati) nei vari server in modo tale che quando ci saranno le richieste di lettura siano ben bilanciate tra i server (come effettuare la distribuzione dipende dall'applicazione). Inoltre, il cambio di richieste possono cambiare nel tempo (anche settimanale). Bisogna bene l'andamento del carico sia in base al tipo delle richieste degli aggregati sia in base al tempo in modo da non lasciare alcuni server sovraccarichi e altri scarichi. **Pattern mining** per capire come distribuire le risorse nei server.

NOTE sullo sharding:

- **auto-sharding:** caratteristica del DBMS (ad esempio tramite una funzione HASH), se effettuato dall'applicazione potrebbe essere visto come un problema dato dell'automatismo del sistema. viene però scaricata la responsabilità dell'applicazione
- l'uso dello sharding è importante per le performance perché può migliorare le operazioni in lettura e scrittura.
 - o Combinando la **replicazione** con il **caching** è possibile migliorare notevolmente le letture a discapito delle scritture. Migliora quindi la robustezza delle letture perché ogni replica va bene, per la scrittura invece bisogna stabilire un server master per le scritture in modo da risolvere i problemi per le inconsistenze.
 - o L'uso dello sharding aiuta a scalare in maniera orizzontale le scritture solo se si aggiungono più server e si fa la distribuzione dei dati attraverso i server. Aiuta la scalabilità delle scritture perché distribuendo su più server i dati, è meno probabile che ci siano due operazioni di scrittura che impegnano lo stesso server. Avendo distribuito i dati su più nodi, le richieste di scrittura è più possibile che vadano su nodi diversi.

Master-slave replication server: repliche dei dati sui nodi. Un nodo è il nodo master (o primario) ed è la risorsa che responsabile della consistenza delle copie dello stesso dato (Nota: non è obbligatorio avere un master per tutti i dati). Un nodo master che ha allocato tutti i dati e poi vengono distribuite le replicate negli altri, il master è quindi responsabile delle **sincronizzazioni** e delle modifiche ai nodi slave (secondari). le letture si possono fare quindi sia sul master che sugli slave (**scalabilità rispetto alle richieste di lettura dei dati**). In questo modo il master è visto un po' come il collo di bottiglia del sistema perché le scritture sono accettate solo dal nodo master. Le letture sugli slave non vengono accettate se non sono ancora stati aggiornati e quindi non sono consistenti. L'aggiunta di slaves permette quindi di aumentare la scalabilità in lettura (**read resilience**) ma rimane comunque il problema delle performance in scrittura. Se dovesse cadere la disponibilità del master per problemi di rete o HW del master, le operazioni di lettura potrebbero continuare ad essere accettate da tutti i nodi slaves in quanto si trovano con i valori aggiornati. Un valore

aggiunto (ex MondoDB lo permette), qualora si voglia aumentare le disponibilità in scrittura (anche solo momentaneamente fintanto che il nodo master non viene “resuscitato”) qualche nodo slaves potrebbe sostituire il master iniziando a sincronizzare il valore dei dati con gli altri nodi. Avere più nodi slaves sincronizzati col master (anche in caso di single server) aiuta dal punto di vista della **tolleranza ai guasti** perché in questo caso il master diventa il punto di debolezza dell’architettura.

Nota: non tutti i DBMS lo permettono ma possono esserci più percorsi di lettura e scrittura di supporto.

Lack in consistency: nelle operazioni su più nodi facendo sia replica che sharding appare un problema di inconsistenza nello span temporale fra l’inizio dell’aggiornamento degli slaves e il completamento dell’aggiornamento che rende il sistema consistente.

Peer-to-peer replication: utilizzo dei nodi in modalità P2P, un nodo può essere master per alcuni nodi e slave per altri quindi la responsabilità di essere “master” viene distribuita tra i nodi, gli slave comunicano tra di loro le operazioni in scrittura che hanno accettato in modo da mantenere le copie del dato nei nodi slave. Tutte le repliche hanno lo stesso peso e tutti i nodi accettano le scritture, la perdita di un nodo non è così catastrofica perché rimangono gli altri peer.

- **Vantaggio:** nell’architettura con i nodi P2P si risolve il problema della centralizzazione del master avendo il rischio della perdita dei dati.
- **Svantaggi:** complicazioni in merito alla consistenza (rallentamento dell’operazione di aggiornamento dei dati), aumentando il numero di nodi aumentano le operazioni in scrittura da fare per rendere consistenti i nodi e quindi è più probabile trovare un nodo che abbia un’immagine “vecchia” del dato. Inoltre, nell’ipotesi che ci siano più master per lo stesso dato, ci potrebbe essere un conflitto **write-write conflict** (ovvero due master che tentano di aggiornare lo stesso dato, magari proveniente da due applicazioni differenti).

Risoluzione del **write-write conflict** nella modalità P2P:

- **approccio pessimistico (massima consistenza):** le repliche vengono coordinate in modo da evitare i conflitti. C’è una garanzia forte da parte del master a discapito del traffico sulla rete per coordinare le scritture (non è richiesta la completa adesione delle repliche ma solo la maggioranza, ovvero si accetta lo stato “scrittura con successo” quando la scrittura viene effettuata sulla maggior parte dei nodi). Un’operazione di scrittura non è dichiarata “terminata con successo” fino a che non viene effettuata con successo sul numero di nodi pari alla maggioranza tra i nodi del cluster.
- **approccio ottimistico (massima disponibilità dei dati):** si accetta le scritture e le si rendono più veloci possibili dichiarando una scrittura completata appena termina la scrittura su un nodo. Operazioni di scrittura più veloci quindi maggiore **availability** ma rischio maggiore di write-write conflict (e quindi di inconsistenza dei dati)

Combinazione di sharding e replication: i nodi possono avere la funzione di master per alcune copie dei dati e da slave per altre, ma ciascuna copia del dato ha un singolo master. Quindi si ha un nodo “autorevole” per ciascuna copia del dato quindi ogni volta che si ha il dubbio sul valore di un dato ci si riferisce al suo “responsabile” ovvero master. La combinazione dello sharding con la replica in una modalità P2P è una modalità comune per i DB NoSQL (ad esempio quelli di tipo colonna che sono portati a fare aggiornamenti frequenti fra le colonne).

Problemi che si hanno per garantire la consistenza del dato:

- **Strong consistency:** i db relazionali gestiscono la consistenza (anche nei db distribuiti) mantenendo più copie del dato in maniera distribuita e utilizzano il **lock** sul dato in modo da evitare più operazioni di scrittura sullo stesso dato. I db relazionali hanno l'obiettivo di mantenere la consistenza di tipo forte in modo da evitare i problemi di inconsistenza a prescindere.
- **Update consistency:** problema di consistenza a fronte di un'operazione di "aggiornamento", avviene perché si hanno due richieste di fare update sullo stesso dato, dove il dato è mantenuto in maniera distribuita con repliche su più server. Il server potrebbe serializzare le due operazioni di scrittura eseguendole sulla stessa copia del dato (prima una o poi l'altra)
- **Failure consistency:** fallimento della richiesta. Date le due operazioni di update, la seconda agisce su un dato che non è consistente rispetto alla prima operazione di update, cioè il secondo aggiornamento è basato su uno stack del dato che non è il risultato della prima scrittura ma quello precedente alla prima scrittura.

Nota:

- **2-face-commit:** si usa nei db relazionali, è un'operazione a due fasi dove le transazioni che accedono ai dati iniziano a chiedere il lock dei dati a su cui vogliono fare la modifica e di tutto ciò che necessitano per completare l'operazione, otterranno il lock e poi rilasceranno la risorsa alle altre iterazioni che si comporteranno allo stesso modo della prima

Approcci a fronte di un problema di consistenza:

- **Ottimistico:** permette alle operazioni di effettuare gli aggiornamenti facendo un test, prima di scrivere sul valore che ha il dato, se il valore è stato modificato rileggerà di nuovo il valore e sulla base di quel valore scrive. Si chiama **conditional approach**, ovvero chiede alle operazioni che devono scrivere di verificare prima delle operazioni di scrittura se il dato nel frattempo è stato modificato da qualche altro client. **conditional update**. Ovvero le operazioni sono: legge il dato, lo elabora e prima di scrivere controlla che il dato sia ancora quello letto precedentemente.
- **Pessimistico:** viene usato ad esempio dei db relazionali che risolvono il problema alla radice e fanno in modo che non si presenti proprio l'inconsistenza. Usano i lock. Aspetto negativo: diminuisce la disponibilità dei dati, quindi il numero di transazioni per secondo è più basso rispetto a quello che si potrebbe avere senza lock.

Sequential consistency: ha lo scopo di risolvere il problema della consistenza in sistemi distribuiti, si assicura che tutti i nodi effettueranno le operazioni nello stesso ordine. Si può parlare di **P2P replication and sequential inconsistency**. Un approccio ottimistico è: salvare le due versioni e controllare a valle se siano corrette.

Optimistic handling from version control systems: approccio in cui si effettua il controllo delle versioni per gestire il **write-write conflict**. Viene effettuato un **merge** delle operazioni di modifica, il problema viene risolto a valle solo quando si presenta e quindi non si cerca di risolvere il problema a monte prevenendolo. Questa è un'ipotesi ottimista che pensa il conflitto sia raro. Ovviamente la politica per fare il merge non è universale ma dipende dal dominio e deve essere programmata da caso a caso.

Tradeoff fra safety e liveness: gli approcci pessimistici, sebbene risolvano il problema evitando il sorgere di conflitti, degradano in maniera severa (forte) da disponibilità e la capacità dell'applicazione sw di rispondere a molte richieste (molto spesso senza motivo). Ad esempio ci potrebbero essere i problemi dei **deadlock**.

Logical consistency: legata a un vincolo applicativo, ovvero una logica operativa che lega il valore di due relazioni diverse tra di loro. Ad esempio, viene effettuata la lettura di un dato durante un'operazione di modifica, il dato risulta inconsistente, è possibile usare le transazioni per rendere atomiche alcune operazioni di update.

Nota: le transazioni non sono disponibili a tutti i NoSQL DB, i db a grafo le forniscono, è possibile quindi includere due scritture sul dato (all'interno di una transazione) e permettere la scrittura a valle al termine della transazione (oppure prima ma non a metà dell'operazione di modifica). Invece, i db che non fanno uso di aggregati, di solito non hanno le transazioni ma si affidano al fatto che i dati del modello aggregato sono disponibili in locale sullo stesso modo e quindi le operazioni che modificano un aggregato avvengono in maniera atomica, però non viene garantita l'atomicità nelle operazioni che coinvolgono più aggregati.

Inconsistency window: intervallo di tempo in cui si sa che i dati non sono consistenti. Se bisogna andare a modificare i dati nello stesso server, questa finestra di inconsistenza potrebbe essere molto piccola (ad esempio Amazon ha una finestra di inconsistenza di meno di 1sec, per Amazon la cosa più importante è rendere a disposizione i dati il più frequentemente possibile per non spazientire il cliente e poi se ci sono dei problemi vengono gestiti a valle). Aumentando il numero di repliche, peggiora la finestra di inconsistenza e in presenza di inconsistenza logica questo problema aumenta ulteriormente perché vuol dire che ci sono più copie dei dati che devono essere allineati su più nodi.

Eventual consistency (consistenza finale, alla fine di una finestra): situazione nella quale alla fine della finestra di inconsistenza, tutte le repliche si troveranno allineate e si avrà la consistenza forte di tutti i dati, prima invece è possibile che alcune copie si troveranno **out of date** (ovvero in uno stato vecchio/sterile). Questa è la garanzia di consistenza che c'è a disposizione in tutte quelle applicazioni che danno la priorità alla disponibilità del dato. Nota: "eventual" in inglese significa "finale".

Read-your-Writes consistency: al seguito di una modifica non è detto che si possa subito leggere la versione finale. Ad esempio, si pubblica un commento, si riavvia il browser ma il cluster non è stato ancora reso consistente quindi c'è l'impressione che il commento non sia stato pubblicato (sensazione che gli aggiornamenti vengano persi perché magari la lettura non viene effettuata nello stesso server che ha accettato l'operazione di scrittura).

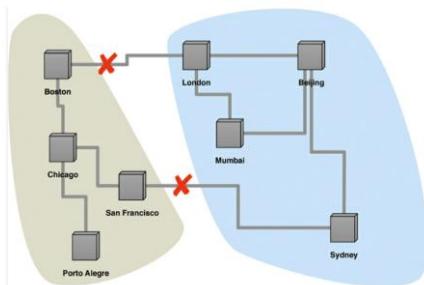
Session consistency: consistenza di sessione implementata da una **sticky session**, punta a garantire la **read-your-write consistency**. Una app client lavora spesso in una sessione.

- Incapsula il lavoro che l'app client fa sui dati all'interno di una sessione utente. Bisogna quindi assegnare un'etichetta che associa l'identificatore di sessione al dato che è stato modificato, così facendo questo prende il nome di **sticky session** (una sessione che rimane appiccicata al dato fino a che la sessione rimane attiva).
- Un altro modo per garantire la **session consistency** è l'utilizzo dei **timestamp** (o numeri di versione), sono un modo per memorizzare una "storia" che è intercorsa sul dato. Ci fossero più server che vogliono modificare lo stesso dato, il dato sarà soggetto a più versioni che ordinano quindi le modifiche che il dato subisce nel tempo. Il nodo server si deve assicurare di avere l'update del record che include la versione del timestamp fornito dalla sessione prima di rispondere alla richiesta, se le versioni non fanno **match**, la modifica è rifiutata dal server (**conditional update**).

Relaxing consistency: la consistenza è una proprietà desiderabile ma in certe applicazioni distribuite potrebbe rallentare le applicazioni stesse perché impone vincoli troppo forti, potrebbe essere quindi necessario rilassare questi vincoli a favore ad esempio dell'**availability** del dato. Bisogna quindi capire quanto si è disposti a rifiutare i vincoli di consistenza forti a favore di altre proprietà (tradeoff fra la consistenza e le altre proprietà, alcune applicazioni possono tollerare inconsistenze).

The CAP Theorem: (Consistency Availability Partitioning), sono tre proprietà desiderabili delle app distribuite. I dati devono essere consistenti nelle repliche, i dati devono essere sempre disponibili (e consistenti) e che l'applicazione sia robusta e capace di sopravvivere a un temporaneo partizionamento della rete. Partizionare vuol dire accettare la situazione nel quale i nodi nel cluster di server che comunicano fra di loro, temporaneamente non potrebbero comunicare tra di loro e quindi una parte della rete rimane "partizionata" (isolata). Il CAP Theorem dice che, date queste tre proprietà, non si possono avere tutte e tre allo stesso momento ma se ne possono avere solo 2/3. Quindi non può essere garantita la terna delle proprietà contemporaneamente, bisogna scegliere a quale delle tre prerogative bisogna rinunciare in funzione dalle necessità dell'applicazione.

Partitions in a cluster



Types of trade-offs among the three options

Types of trade-offs (default)	Databases	Comments
Consistent and available (CA)	Redis, PostgreSQL, Neo4J	They do not distribute the data and they do not have the partitions
Consistent and partition tolerant (CP)	MongoDB, HBase	The hardware faults are managed disabling a partition and allowing the active nodes to answer instead of the missing node
Available and partition tolerant (AP)	CouchDB	It does not guarantee the consistency of data between the two servers

Nota: la tabella si ispira a "6 db in 6 weeks". Naturalmente i tipi di trade-off dei db possono cambiare nel tempo, non sono fissi.

Partition tolerance:

- un single-server è sempre un sistema CA: non ha partition tolerance
- in un cluster è difficile usare un sistema CA perché vorrebbe dire occupare una buona parte della rete per fare request tipo "ci sei? puoi rispondere? ecc". Ogni nodo andrà in down durante le operazioni per assicurare la consistenza.

- i cluster devono essere quindi tolleranti rispetto al partizionamento della rete, rimane quindi il trade-off tra Consistency e Availability. Nota: non è una decisione binaria ma è possibile garantire le due proprietà in maniera più o meno alta.

Trade off tra Availability e Latency: ovvero tra la disponibilità e il tempo di risposta. Invece che le proprietà ACID, i db NoSQL si dice che seguano le **proprietà BASE (Basically Available, Soft state, Eventual consistency)**.

Trade-off tra Consistency e Latency: se si è disposti a tollerare la latenza allora allora si possono dare chance maggiori alla consistenza. È possibile aumentare la consistenza avendo più nodi coinvolti nell'interazione ma ogni nodo aggiunto al cluster aumenta il tempo di latenza.

Relaxing durability: rilassamento della durabilità (per durabilità di intende la persistenza del dato).

- La durabilità è un'altra delle caratteristiche che ci si aspetta vengano garantite dal DBMS. Tuttavia, questa è un'altra caratteristica che si potrebbe voler rilassare, ovvero far venir meno la persistenza del dato. Alcune volte la durabilità non è così indispensabile per la natura stessa del dato o perché i dati non verrebbero comunque persi, è possibile quindi rinunciare la durabilità ma aumentare altre caratteristiche come la disponibilità e la velocità dell'applicazione.
- Un altro esempio è salvare lo stato della sessione utente. Supponendo di avere un grande sito web, un'app utente e un'applicazione che gestisce l'iterazione dell'utente con il sito, si suppone che si memorizzi lo stato della sessione in memoria (qualche volta potrebbe capitare che il DBMS perda la persistenza del dato ma tutto sommato non è un qualcosa di troppo problematico perché l'utente si scoccia ma può pur sempre avviare una nuova sessione, capita però raramente ma aumentano i tempi di risposta del sito web quindi l'utente risulta più soddisfatto comunque).

Lack of durability in replication: un altro trade-off in cui la durabilità è coinvolta è nel caso della replicazione.

- Un fallimento della durabilità nella replicazione si presenta quando un nodo processa una aggiornamento ma fallisce prima che l'update sia replicato negli altri nodi.
- In un modello di distribuzione master-slave, si presenta quando uno slave nomina automaticamente un nuovo master in caso di errore del master esistente. Quando un master è in fail, tutte le scritture non trasferite sulle repliche verranno perse perché si presenteranno dei conflitti fra le copie del vecchio master e del nuovo.
- Un miglioramento per garantire la durabilità della replicazione è porre il master in attesa che più repliche riconoscano l'update e quindi solo dopo lo riconosce al client.

Quorums: sono un modo pratico per utilizzar ei nodi a disposizione. Si ha un cluster con un certo numero di nodi e il fattore di replica (ovvero il numero delle copie che si tengono per ciascun dato). Il numero di server del cluster è generalmente molto superiore rispetto al fattore di replica che è generalmente un numero intero non molto alto e possibilmente dispari (perché si vuole avere una maggioranza di nodi in accordo su una versione del dato in un asso di tempo, ci si riferisce alle repliche).

- **Write quorum:** si avrà poi meno possibilità di leggere un dato vecchio. Si stanno un po' ostacolando le operazioni di scrittura, quindi la disponibilità del dato, a favore di una consistenza più forte. La formula $W > N/2$ dice che nel momento che bisogna fare una scrittura, il DBMS può rispondere "scrittura effettuata con successo" quando sono state scritte più di $N/2$ copie del dato quindi quando ci saranno delle operazioni di lettura sarà facile rilevare un'inconsistenza perché non ci può essere una maggioranza di dati non aggiornati.

W = quorum in scrittura, N = fattore di replica.

- Imagine some data replicated over three nodes ($N=3$).
- You don't need all nodes to acknowledge a write to ensure strong consistency; all you need is 2—the majority of the N nodes.
- If you have conflicting writes, only one write can get a majority. This is referred to as a write quorum and expressed by the inequality of:
- $W > N/2$,

meaning the number of nodes participating in the write (W) must be more than the half the number of nodes involved in replication (N).

- The number of replicas (N) is often called the replication factor.

Read quorum:

W = quorum in scrittura, N = fattore di replica, R = quorum in lettura.

- How many nodes you need to contact to be sure you have the most up-to-date version of your data?
- The read quorum depends on how many nodes need to confirm a write (W).
- Let's consider a replication factor (N) of 3.
- If all writes need two nodes to confirm ($W = 2$), then we need to contact at least two nodes to be sure we'll get the latest data. (2 if the two read copies agree; 3 if not).
- If at a given time, writes are only confirmed by a single node ($W = 1$), we do not have a strong consistency on writes, and we have to detect a possible write conflict. We need to talk to enough readers (all three nodes) to be sure we have the latest updates.
- Thus we can get strongly consistent reads even if we do not have a strong consistency in writes.
- The number of nodes (R) that you need to contact for a consistent read, is: $R + W > N$ with W the nodes confirming a write, N the replication factor.

Nota: R e W possono cambiare nell'applicazione in funzione di quanto sono importanti le letture e le scritture, cioè non sono univocamente determinati ma dinamicamente determinati dall'applicazione stessa. Ex $W=3$ e $R=1$ (viene avvantaggiata la lettura che può leggere da un solo server a discapito della scrittura), $W=1$ e $R=3$ (avvantaggiata la scrittura ma non la lettura).

Version stamps: sono associati al dato e vengono restituiti all'applicazione insieme al dato stesso, permettono. Ad esempio, non si possono bloccare i dati che servono in un form utente fino a che l'utente non lo completa e preme "submit". È una **offline concurrency (Optimistic Offline Lock)**, una concorrenza gestita in maniera offline, cioè è una situazione con approccio di tipo ottimistico nel quale ci si compatta con un **conditional update** (si blocca il dato e si comunica che si vuole fare una scrittura solo se il **version stamp**, che si ottiene da sistema nel momento che si cerca di scrivere, è rimasto inalterato rispetto al **version stamp** che si aveva all'inizio dell'operazione. Se i due **version stamp** sono sincronizzati vuol dire che si sta scrivendo il valore del dato che è rimasto inalterato, il dato non è stato quindi bloccato e le operazioni di verifica

avvengono a valle. Se non sono sincronizzati viene rifiutata l'operazione perché il dato non è più consistente. Ad esempio, in HTTP c'è il tag chiamato **etags** che dice lo stato della risorsa, quindi la versione attuale della risorsa.

Tipi di version stamp:

1. You can use a counter, always incrementing it when you update the resource.
Counters are useful since they make it easy to tell if one version is more recent than another.
On the other hand, they require the server to generate the counter value, and also need a single master to ensure the counters aren't duplicated.
2. Another approach is to create a GUID, a large random number that's guaranteed to be unique (a combination of dates, hardware information, etc).
The nice thing about GUIDs is that they can be generated by anyone and you'll never get a duplicate.
A disadvantage is that they are large and can't be compared directly for recentness.
3. make a hash of the contents of the resource. With a big enough hash key size, a content hash can be globally unique like a GUID and can also be generated by anyone; the advantage is that they are deterministic but cannot be compared for recentness
4. use the timestamp of the last update. Like counters, they are reasonably short and can be directly compared for recentness. Yet have the advantage of not needing a single master. Multiple machines can generate timestamps—but to work properly, their clocks have to be kept in sync.
A disadvantage is that one node with a bad clock can cause all sorts of data corruptions. There's also a danger that if the timestamp is too granular you can get duplicates—it's no good using timestamps of a millisecond precision if you get many updates per millisecond.

Blending different schemas: mischiare le diverse tipologie di version stamp per ottenere differenti vantaggi.

Version stamp e P2P: la versione a contatore può funzionare solo nei sistemi a single-server (un master e tutti gli slave si riversano su di lui), in caso contrario potrebbero presentarsi dei potenziali conflitti.

Vector stamp: i NoSQL li utilizzano, sono vettori che mantengono i noti che possiedono le replicate, ognuno di questi hanno un contatore differente. Quando due nodi si mettono in comunicazione, si sincronizzeranno (dati e vector state). C'è una visione complessiva della situazione. La modalità con cui i noti si sincronizzano dipende da schema a schema (non è fisso).

- By using this scheme you can tell if one version stamp is newer than another because the newer stamp will have all its counters greater than or equal to those in the older stamp.
- So [blue: 1, green: 2, black: 5] is newer than [blue:1, green: 1, black 5] since one of its counters is greater.
- If both stamps have a counter greater than the other, e.g. [blue: 1, green: 2, black: 5] and [blue: 2, green: 1, black: 5], then you have a write-write conflict.

Introduzione al paradigma di Map Reduce

MapReduce: è un paradigma di computazione distribuito. È adatto a trattare grossi volumi di dati in maniera distribuita (quindi per funzionare tanti server in rete tra loro).

It was proposed by Google in MapReduce: Simplified Data Processing on Large Clusters, by Jeffrey Dean and Sanjay Ghemawat in OSDI 2004.

È diventato un tool per eseguire queries in **datastores partition-tolerant**. Permette la divisione dei task in piccole componenti da eseguire in maniera concorrente sui server del cluster.

MapReduce suddivide i problemi in due parti:

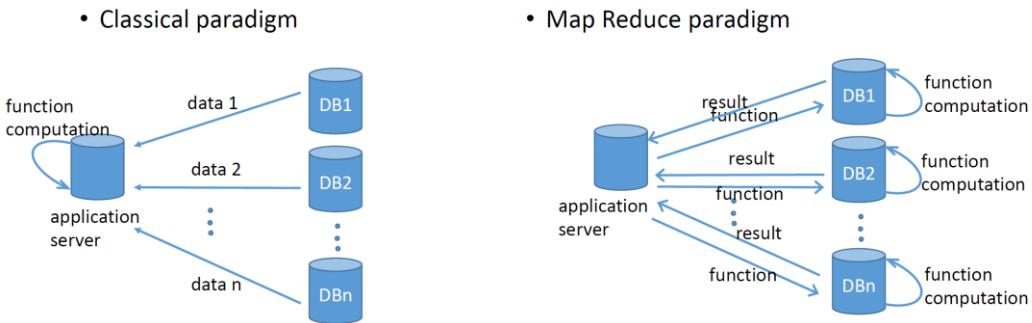
- Converts a list of data on the same cluster node in another list by means of the application of the function **map()**; **map()** maps each object according to a key and groups the objects for the common key. Ovvero è una conversione da una lista a un'altra lista in maniera tale che la conversione avvenga in locale nello stesso nodo, quindi senza necessità di dover spostare i dati da un server a un altro (è il principio ispiratore di MapReduce: far eseguire in locale la computazione in modo da non far trasferire troppi dati sulla rete da un server ad un altro). Una di queste componenti è **map()** che prende la lista in ingresso come parametro, ha una chiave di riferimento per gli oggetti che vengono trattati e in base a quella chiave trasforma gli oggetti in una nuova lista che viene restituita in output.
- Converts the second list, generated by **map()** in one or more scalar values by means of the application of the function **reduce()** (a sort of aggregate function) **reduce()** reduces to a single value the objects grouped for the common key; if there are multiple results of the map functions for the same key executed on different nodes, they are aggregated into one. In questa parte c'è la funzione **reduce** che ha lo scopo di ridurre il volume dei dati (tipicamente facendo delle statistiche sulle chiavi, non è detto che la chiave utilizzata per l'elaborazione statistica sia la stessa usata in **map**).

Ad esempio: (?) **map** estrae da un testo tutte le parole (chiavi) e **reduce** calcola la frequenza di una parola nel testo.

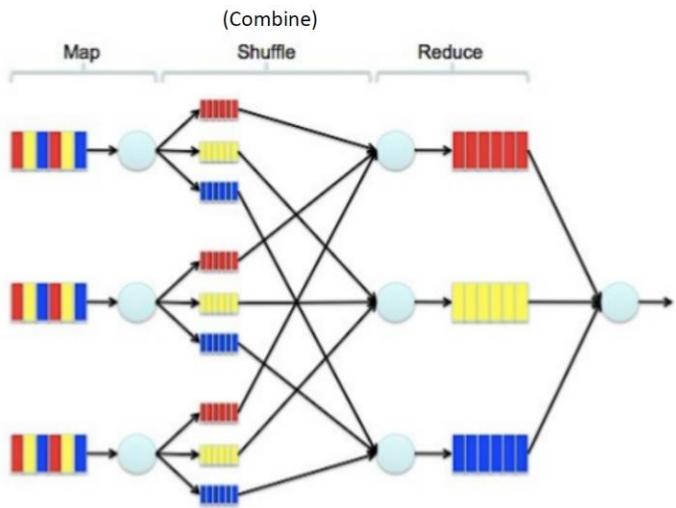
Paradigma di programmazione funzionale: non richiede un grande spazio per la memorizzazione delle variabili perché non è necessario salvarsi lo stato locale.

Paradigma MapReduce: su grossi volumi di dati, il paradigma classico non è possibile perché vuol dire non "dare la rete di dati" ad ogni necessità di calcolo. Il paradigma di MapReduce, piuttosto che trasferire i dati dai DB server all'application server, trasferisce le funzioni (**map** e **reduce**) dal server applicativo ai vari nodi che tengono i dati e la funzione viene calcolata in locale dove riedono i dati senza dover necessariamente trasferire enormi quantità di dati nella rete, il risultato viene quindi mandato all'application server che eventualmente effettua il **reduce**. Quindi si cerca di fare la maggior parte dell'elaborazione dei dati in locale dove sono stati memorizzati (princípio ispiratore).

È stato quindi applicato lo sharding, sono stati distribuiti i dati su più nodi del cluster ed è stata applicata la funzione di elaborazione di dati su ciascun nodo in modo che questa funzione lavori in locale. I risultati, ridotti in cardinalità il più possibile, vengono poi spediti all'application server che poi fa le ultime elaborazioni in modo da occupare meno traffico possibile sulla rete.



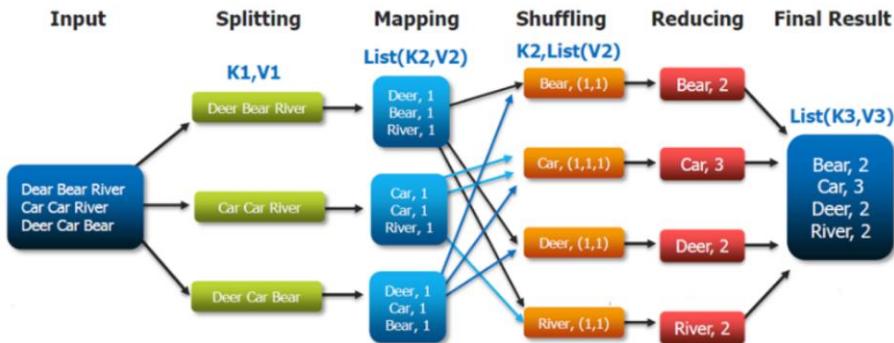
MapReduce, different implementations: ci sono diversi schemi implementativi del paradigma MapReduce, modo popolare è il progetto di **Apache Foundation Hadoop** che ha anche operazioni aggiuntive rispetto a map e reduce. Molti NoSQL DB hanno delle query in cui possono specificare le funzioni di map e reduce come elaborazione dei dati interrogati nella query.



Note:

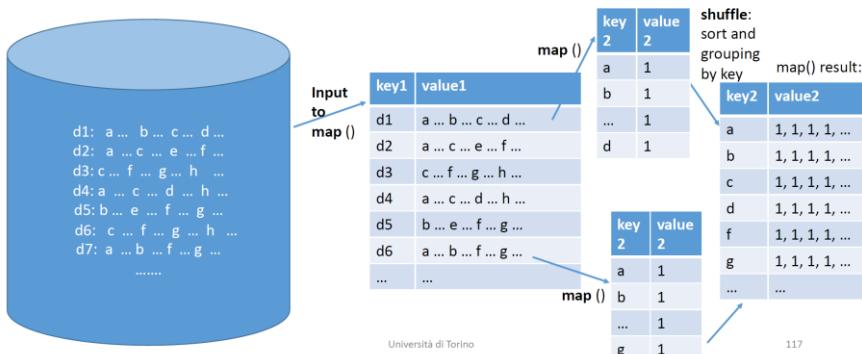
- L'immagine usa i colori per rappresentare le chiavi
- I grossi volumi di dati sono stati già divisi in tre nodi nei quali si hanno diversi dati allocati.
- La funzione di map raggruppa per chiave tutti i valori che sono stati allocati su un nodo e prende in input la lista dei valori e produce delle liste di valori ciascuno con la stessa chiave, lo stesso valore di chiave è associato con lo stesso valore. La map è stata mandata in locale a ciascuno dei nodi e fa lo stesso lavoro con ciascuno dei dati allocati in ciascuno dei nodi. Il risultato della funzione map può essere spedito così com'è attraverso la rete verso i nodi in cui si esegue la seconda funzione di reduce, la quale applica ai valori associati alle stesse chiavi una funzione analitica (ad esempio conta ciascun valore che restituisce un valore scalare come risultato della funzione aggregata).
- La fase di **shuffling** (o **combining**) è la fase di “re-direzione” che ridirige le liste dei valori con stessa chiave, residenti nei diversi nodi, verso uno stesso nodo che sarà il concentratore di tutti i valori associati a una stessa chiave. Lo shuffling riordina e ridistribuisce il carico di lavoro per la reduce su nodi diversi in base alle chiavi.
- Fisicamente la reduce può essere eseguita sullo stesso nodo oppure anche distribuita su più nodi.

Esempio di calcolo della frequenza delle parole nei documenti (applicato da Google):



Detail of the example: map() step

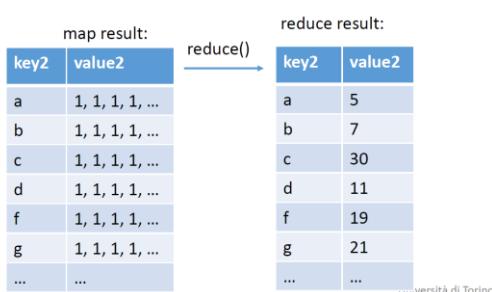
- transforms the input (a map *key1-value1*) into another map (*key2, value2*) where *keys* and *values* are arbitrarily chosen by the programmer



Nota: Key1 è l'ID del file.

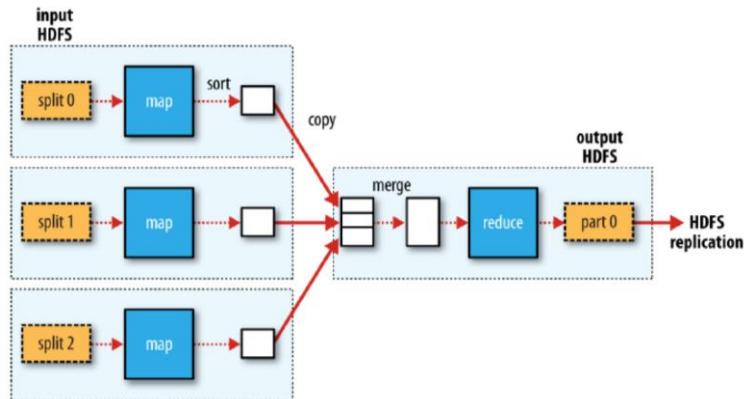
Detail of the example: reduce() step

- produces from the result of map() a second map (of *key2-value2*) where the sequence of the values associated to the same key have been aggregated in a unique result (scalar)



Hadoop, come implementa il MapReduce: ha una prima funzione di splitting che serve per separare l'input in task diversi che vengono poi affrontati su nodi diversi. C'è l'operazione di map (che non fa altro che elaborare l'input e allocarlo in memoria, separando chiave e valore dall'input), poi l'operazione di sorting per

far emergere anche la presenza della chiave con un valore più volte in maniera consecutiva nella lista in modo che questa lista venga poi copiata sulla rete in un altro nodo dove viene poi applicata la reduce. Nel nuovo nodo c'è un'operazione di merge (se c'è la stessa chiave su nodi diversi, questi vengono messi uno consecutivo all'altro nella lista che viene poi data in input a reduce la quale produce un risultato finale che viene detto "part", ovvero un risultato partizionato -> ci possono essere più part: part1, part2, ...). Il FS di Hadoop è HDFS, ovvero un FS distribuito e ad hoc per il progetto.



Alcune applicazioni in cui il MapReduce è stato applicato (Hadoop): sono classiche applicazioni su grandi volumi di dati

- Risk modeling
 - Customer churn analysis: evitare che i clienti vadano dalla concorrenza
 - Recommendation engine
 - Ad targeting
 - Point of sale transactional Analysis
 - Threat analysis
 - Trade surveillance
 - Search quality
 - Data Sandbox

Example

- On the documents example, we could devise a MapReduce schema with the goal of counting the number of documents that contain a certain word ("mum")
- Instead of passing the documents to a server (that applies a function for the word search in the documents) we pass the function to the servers that work separately, without moving the documents

```

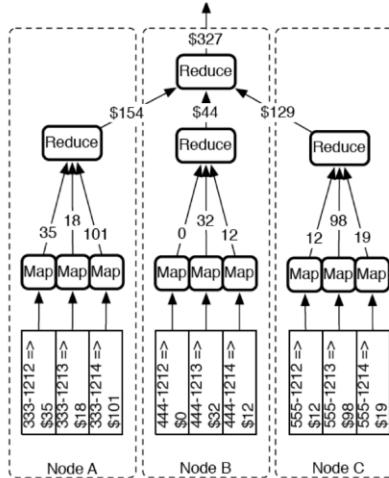
    iterate on any elements d in a list
    and produces another list, given in input to reduce
• map = function(document d, word w) {
  if find(w,d) {return 1} else {return 0};
}

    output list
    function
• list_values_mapped = map(d.find(w), d in
  document_list) → Input list

Scalar value      function      Computed value
• cont = reduce(lambda a,v: a+v, v in
  values_mapped, 0) → Input list
    Initial value of the accumulator a
}
    
```

Example of application:

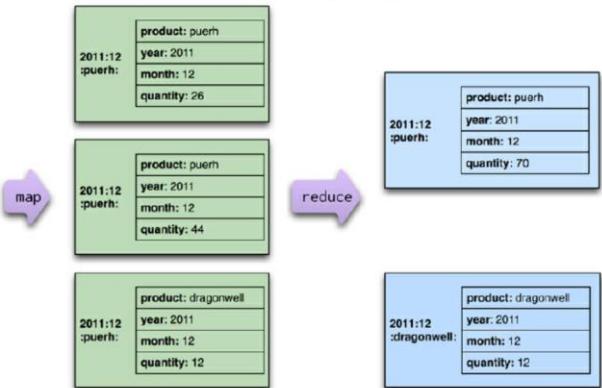
- Computation of the total due by the clients of a telecom company
- The key is the telephone number
- The clients are spread to the server according to the prefix
- Map* applies the filter function that selects the value of the sum to be payed and passes it to *Reduce*
- Reduce* aggregates the values and moves the result to a further *Reduce* step that aggregates them again



Nota: nell'esempio sopra, la somma viene effettuata su risultati già aggregati e quindi a somma degli importi non si effettua sul totale ma sulle somme delle regioni che hanno calcolato la somma degli importi al loro interno in locale. Non è un'operazione che è sempre fattibile, la funzione "somma" però gode della proprietà distributiva e quindi la si può distribuire su più nodi e poi riapplicare sui totali parziali (la funzione media invece non gode di questa proprietà se non con dei "trucchetti").

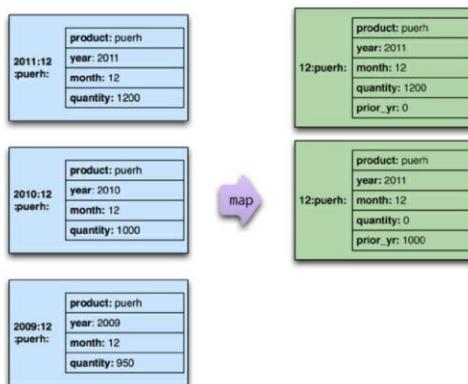
Two stages of Map Reduce:

First stage: reading input records



Second stage: processing the output according to the year

The second stage mapper creates base records for year-on-year comparisons



Incremental Map Reduce: le map-reduce a due fasi possono essere pensate in maniera incrementale dove la funzione di map esegue solo in quei task dove ci sono dei dati nuovi e la funzione reduce ingloba nei dati precedenti, altrimenti la funzione map non viene scatenata. I reducer sono “combinabili” potendo quindi riutilizzare le funzioni (bisogna fare attenzione che la natura delle operazioni lo permettono).

Polyglot persistence: “persistenza poliglotta”, ovvero che usa diversi modelli dei dati con riferimento a diversi tipi di DBs (scegliendo di volta in volta il modello dei dati più adatto al tipo di applicazione che si vuole implementare, questo migliora la produttività del programmatore perché migliora il mismatch di inferenza tra il modello dei dati nel programma e quello che usa il DB per rendere persistenti i dati).

Vantaggi: non c’è il vincolo di dover utilizzare un unico modello che potrebbe portare complicatezze in alcuni casi.

Svantaggi: costi di gestione nell’apprendimento di una tecnologia eterogenea. Con molteplici DBs potrebbe essere che ci siano molteplici server da gestire con problematiche diverse.

This is a pivotal time in databases

- For years the relational model has been a *de facto* option for any type of problems
- In these last years, developers are realising that alternative options exist, collectively known as NoSQL databases that provide features like:
 - schemaless data structures
 - simple replication
 - high availability
 - horizontal scaling
 - new query methods
- We are learning the various functionalities and trade-offs provided by the different NoSQL databases alternatives (e.g., durability vs speed, absolute vs eventual consistency) and will learn to make the best decisions for each use case.

Each problem has different needs

- Given a certain problem to solve, we have to ask ourselves which database or combination of, is most suitable to solve the problem
- Each database corresponds to a category of data model
 - *Relational*
 - *key-value*
 - *columnar*
 - *document-oriented (JSON)*
 - *Graph*
- We have a wide variety of databases that originally were created to answer to different issues

Possible needs

- In order to understand which is the correct data model (and corresponding database) we have to list the application needs and the available resources:
 - Flexible queries on arbitrary attributes
 - indexing for quick data retrieval
 - ad hoc queries vs. planned queries
 - Rigid schema vs. renegotiable schema
 - Data partitioning; replicas (copies); data distribution with hashing
 - Database tuning for reading/writing operations
 - Scalability - horizontal (more servers), vertical (additional computational power to given servers) or mixed

Databases and needs

Database type	Needs, goal, requirement	Examples
Relational	Query flexibility (SQL); easy retrieval of data with join and views (set theory); Data have a predefined type	Oracle, PostgreSQL, Mysql
Key - value	Map each key with a value in a similar way as with a programming map or with a hash table with a possible iteration on the keys; caching in main memory of objects for dynamic web applications	Memcached (memcachedb, membase) Voldemort, Redis, Riak, Amazon Dynamo
Columnar	Store big data volumes on different servers (nodes); Easy in horizontal scalability (addition of servers and columns); Data sparsity; the relationships between columns have minor importance; Guarantee of consistency	Hbase, Cassandra, Hypertable, Google BigTable
Document-based	A document consists in a field ID associated to values of various type such as hash; can contain nested structures; There exist some restrictions due to the document representation	MongoDB, CouchDB
Graph-based	Data are highly interconnected in a graph, in which both nodes and edges might have a property (key-value); ease in browsing the graph	Neo4J 140

Nota: quando ci si trova in situazioni in cui si vuole un DB che permetta di fare statistiche su grossi volumi di dati sparso, dove è eterogeneo lo schema dei record, allora si va su un DB colonna.

Disadvantages of the *Polyglot persistence*

- *Polyglot persistence* has a cost in complexity:
 - Each storage mechanism introduces a new interface
 - The new technology needs to be understood well to allow us to obtain good performances
 - Many offers from the NoSQL community need the execution on big clusters of PCs. Therefore there are issues related to consistency and availability of data. The transactional context, used in the last years as a guarantee of the data validity, does not hold anymore.
- The change in paradigm will not be fast – the software development companies are conservative by nature.

Advantages of the *Polyglot persistence*

- When relational databases are used for inappropriate cases, they provoke a significant increase in the complexity of the software development
 - Let us take for example the case of a software application that is the backend of web pages and is used only to serve them.
The application essentially needs to create the ID of the pages elements; it does not need the transactional context whose goal was originally the data sharing with the database. This is a kind of problem that typically needs a *store of key-values* rather than the corporate traditional relational database.
A good example of the right choice and usage of databases is *The Guardian* – it experimented a sudden increase in productivity with MongoDB instead of a relational database.
- Another benefit comes with the execution on a cluster of *commodity servers*.
The cluster enables the traffic scalability in other ways than the vertical one (increasing the computational power of a single server).
Many NoSQL databases operate on clusters and are able to manage higher traffic volumes than a single server: this is the typical case of big data.
 - Let us take for example the case of the Danish health-care system in which the data access has the goal of obtaining the whole list of prescriptions for a single patient – it makes a data access operation an aggregation operation. The system naturally adopts event programming that is used to populate both the databases, MySQL and Riak (key-value).
Riak manages the load of the update operations instead of MySQL.

Riak: un data store di tipo chiave-valore

È un DB di tipo key-value che può essere interrogato tramite REST (quindi interrogazioni http).

- + The value can be anything – text, JSON, XML, images, video clips etc.
- + It is tolerant to faults to servers (that are cluster nodes)
- + The queries are done on the web, via URLs, header, ecc.
- + The answers are done via HTTP
- + It lacks the support for ad-hoc queries
- + It is suitable to provide support to data centers that need a low latency in the answer and is suitable to support the traffic growth to web pages
- It is difficult to perform the logical links among the values (it does not have foreign keys).
- Query rigide e limitate. È possibile usare solo le chiamate REST
- + Bassa latenza in risposta.

Querying a cluster node:

- The cluster of the servers/nodes are arranged as in a ring with all nodes that are peers
- Any node can be queried via REST (REpresentational State Transfer) by means of the commands: POST (create), GET (read), PUT (update), DELETE and receives information in the HTTP header.
- The URL that we have to contact in order to query the database has the format:
`http://SERVER:PORT/riak/BUCKET/KEY`
- Riak divides the classes of keys to be stored in the database in **buckets** in order to avoid collisions between keys

Note:

- I bucket servono per organizzare gli oggetti in classi di chiavi diverse (si possono descrivere come “classe delle chiavi”).
- **Links:** si usano per collegare dati diversi già memorizzati sottoforma di key-value dentro al sistema. Sono gestiti come un path. I links sono unidirezionali. “riaktag” è il tag che descrive il link. I links possono puntare anche a buckets diversi.
- **“mapred”:** è una parola chiave per indicare “map reduce”
- **Riak** ha un’architettura ad anello, va bene per essere eseguito su tanti nodi di tipo P2P (non ci sono nodi master-slave).
- Nei **VNODE** si hanno delle partizioni di chiavi corrispondenti agli oggetti che si andranno ad utilizzare.
- **Riak** non necessariamente garantisce la consistenza forte durante gli updates, sta al compilatore settare i valori di W e N a seconda delle proprie esigenze. N corrisponde al numero al numero di scritture eseguire con successo per poter dire di aver completate le operazioni di scrittura.
- **Vector clock:** contiene il numero delle versioni. In ogni istante di tempo è una sorta di “immagine globale” che dice in questo momento, per ogni oggetto, qual è il numero di versione che il client mantiene per quell’oggetto.
- Essendo documenti di tipo key-value, molto spesso i documenti contengono valori di tipo testuale (indicizzati da strumenti di information retrieval a partire dalle parole) e quindi potrebbe essere necessaria una ricerca testuale.
- **“strumenti di information retrieval”:** supporto per la ricerca testuale. Molto spesso gestiscono degli inverted-index (indici inversi) che per ogni parola di ricerca, restituiscono i documenti che contengono quella parola. In Riak possono essere introdotte utilizzando Javascript oppure Erlang. È una funzione di tipo “pre-commit”.
- **Apache Solr queries:** usa la notazione polacca inversa
- Riak, avendo un’architettura P2P, non ha un singolo punto di debolezza (come poteva capitare nei sistemi master-slave)
- Tuttavia, le interrogazioni non sono ancora semplici, robuste e flessibili quindi non sono facilmente gestibili.

[HBase: database di tipo colonnare, Open Source di Apache Foundation](#)

È un database concepito per gestire grosse applicazioni con tanti nodi nel cluster. Non si può installare HBase se non ci sono almeno 5 nodi nel cluster, è quindi pensato per essere altamente scalabile per gestire tantissimi nodi. Inoltre, garantisce una garanzia forte sulla consistenza dei dati. Assomiglia un DB relazionale perché si parla di righe e colonne ma in realtà le tabelle non sono fatte come nel modello relazionale:

- le righe non sono come i record nel modello relazionale e
- le colonne non sono rigide e obbligatoriamente presenti per ogni riga ma vengono istanziate di volta in volta a seconda delle necessità,

Lo schema c’è ma è solo informativo, NON è vincolante. Inoltre, ha ulteriori caratteristiche come:

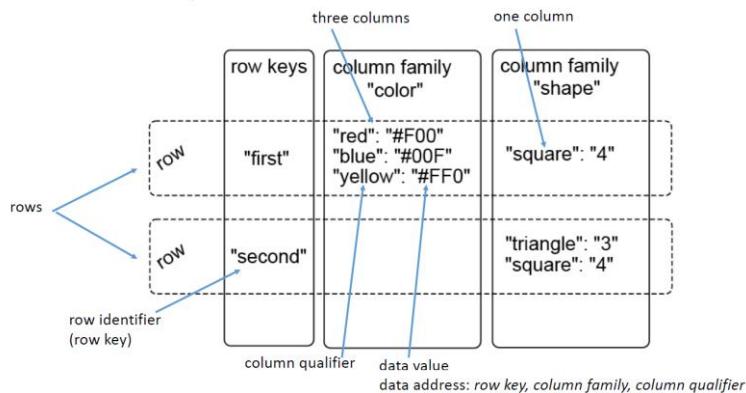
- la capacità di fare versionamento degli oggetti
- capacità di effettuare la compressione dei dati per occupare meno spazio
- gestione del garbage e le tabelle presenti in memoria
- capacità di fare molto efficientemente delle statistiche sui contenuti delle colonne su grandi volumi di dati

È stato inizialmente proposto ispirandosi a BigTable di Google che a sua volta era stato concepito per fare velocemente Natural Language Processing. Avendo così tanti nodi deve essere tollerante ai guasti nel cluster. Gestisce in maniera distribuita le scritture sui nodi e permette una scrittura **write-ahead**, ovvero da una risposta alla scrittura prima ancora che le scritture vengano rese persistenti a livello HW. È stato adoperato da tantissime compagnie ".com" come FB, Twitter, eBay ecc.

Note:

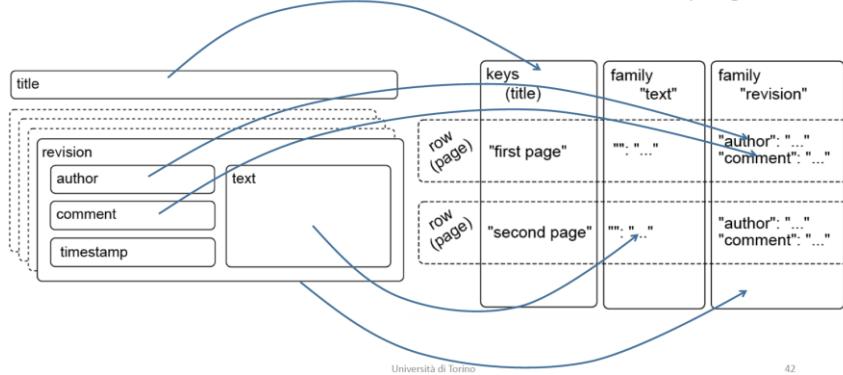
- Per introdurre qualunque tipo di oggetto bisogna introdurre una mappa di tipo key-value, per cui il concetto di tabella è visto come una grande mappa dove le chiavi sono gli identificatori delle righe e il valore è una mappa annidata.
- La mappa annidata servirà per allocare, in maniera flessibile, le colonne che si vorrà allocare per quella singola riga (non necessariamente coincideranno con quelle nelle altre righe).
- I valori delle colonne che vengono allocati nella mappa annidata, non sono interpretati ma sono gestiti come degli array di byte.
- Ogni volta che si introduce una colonna in una riga, si può specificare meglio di che colonna si tratta introducendo una famiglia di colonne. Si possono quindi introdurre diverse colonne ma organizzarle in famiglie. Queste colonne non sono vincolanti. Vengono chiamate **column qualifier**, quindi la specifica delle colonne è rappresentata nel formato **family_name:column_qualifier** dove family_name non è obbligatoria ma volendo è un modo per raggruppare le colonne, columnQualifier è il vero e proprio nome della colonna.

Rows, keys, families, columns, values



- si utilizzano i comandi HTTP di CURL.
- in questo DB, ogni oggetto viene versionato in maniera nativa dal db stesso e tiene tutte le versioni di ogni oggetto (usa il timestamp che rappresenta un numero molto grande di tipo intero).
- Ad esempio, FB usa HBase per gestire i messaggi che si scambiano gli utenti. Lo userID dell'utente è la chiave di riga e il timestamp del messaggio, nel momento in cui si crea il messaggio, viene usato come ID della famiglia di colonne dove le varie colonne di quella famiglia sono le parole contenute nel messaggio. Questo perché le varie colonne vengono singolarmente indicizzate da sistema e così usa gli indici di sistema per andare a ricercare i messaggi in base alle parole contenute nei messaggi. La ragione per cui probabilmente FB usa questo DB è perché sfrutta la sua nativa capacità di indicizzare ogni versione dei dati.
- Wikipedia è un progetto collaborativo aperto, il contenuto di una pagina potrebbe essere versionato (mantenimento della storia dei contributi che vengono dati a una certa pagina) per cui è stata aggiunta una seconda famiglia che è la **revisione** in cui si memorizzano i metadati del contenuto del testo, cioè l'autore e un eventuale commento che giustifica la revisione.

The table with the revisions of the wiki pages



- Operazioni per permettere il versionamento:

- disable 'wiki'
 - Request of an unlimited number of revisions (by default they are only 3):
 - alter 'wiki', { NAME => 'text', VERSIONS => hbase* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
- Denotes a continuation line of the previous one
- Request of addition of a new column family ('revision'):
 - alter 'wiki', { NAME => 'revision', VERSIONS => hbase* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
 - enable 'wiki'

- **jbyte** funzione che converte in array di bytes com'è richiesto da HBase.

- Dato che il volume di tutte queste pagine dell'intera Wikipedia è molto elevato, si propone di comprimere il contenuto di questi testi (questa è un'altra **facility** nativa del DB). È possibile prendere la colonna "testo" e comprimerla secondo ad esempio GZ. È possibile specificare di usare un BOOMFILTER per andare velocemente a sapere se un certo contenuto è stato già inserito oppure no. Questo BOOMFILTER lo si può abilitare a livello di riga o di colonna usando la parola chiave ROWCOL. Questo abiliterebbe una ricerca rapida su una certa colonna se da qualche parte nel sistema in quella colonna è già stato inserito un certo valore.

Bloom filters: sono una variante di una funzione hash. Sono stati proposti nel 1970 da Bloom come un modo per velocizzare le operazioni di **spell-checking** perché quando si va a scrivere si può controllare velocemente in un DB se quella parola è già stata inserita, in quel caso la si va a correggere o tenere così com'è (tipo in vocabolario e il correttore).

Si può pensare di allocare un lunghissimo array di bit inizializzato a 0, quando arriva un contenuto che si vuole andare ad aggiungere nella base dati si calcola sulla base di questo contenuto una chiave hash che sarà lunga quanto l'array di bit (sequenza di 0/1), si vanno ad aggiungere gli 1 nei punti in cui i bit sono pari a 1. Quando successivamente si va a cercare nuovamente quella parola, per vedere se è già stata inserita in passato, si prende la parola e si applica una funzione hash che produrrà questo array di bit e si va a verificare nei punti dove l'array di bit è a 1 se nella memoria quel bit è pari a 1:

- se sì, quella parola è già stata inserita in passato
- in caso contrario vuol dire che quella parola non è ancora stata aggiunta

Sono operazioni molto rapide come un calcolo di una funzione hash in un semplice match di array di bit sui valori 1.

Problema: questo metodo potrebbe dare dei falsi positivi (collisione derivate dalle funzioni hash). La filosofia del filtri Bloom afferma che questo non è un problema perché dato il valore 1, si va a cercare il valore del db e se non c'è si riconosce il falso positivo ma almeno si può sapere se c'è o non c'è da una prima risposta del filtro.

Nota: la funzione hash è sempre deterministica, restituirà sempre la stessa chiave

Bloom Filters vs Hash Tables: le tabelle hash NON danno falsi positivi ma occupano uno spazio parecchio più grande mentre i filtri Bloom vanno a occupare soltanto quella memoria che è stata richiesta di allocare conseguente alla lunghezza degli array di bit.

Region and keys partitioning: organizzare lo spazio disco in modo da ricordare dove sono state inserite le varie informazioni. Ci sono delle tabelle, una sorta di catalogo che il sistema mantiene, chiamate “.META” che serve per ricordare in quale porzione del DB sono state memorizzate le varie chiavi. Il sistema popola una regione del DB con le chiavi consecutive, quando si riempie scrive nella tabella “.META” fino a che chiave è andato a popolare una certa regione e poi passa a quella successiva. Quindi mantiene dei range di chiavi delle righe utilizzate.

Table for the links: ad esempio per Wiki, una collezione di chiavi che permettono la navigazione ipertestuale della piattaforma. È formato da una coppia “to-from”.

Development of a client application with communication protocols: aspetto più che altro applicativo e meno di data model.

- Abilitare un'applicazione client a comunicare col server tramite un protocollo di comunicazione. Un protocollo è Thrift.

Whirr (a tool for the clusters on the cloud): il mondo sta andando verso i sistemi cloud. Questo progetto permette a delle facilities grafiche di lavorare con altri progetti e db come Cassandra, HBase, Hadoop ecc.

Vantaggi di HBase:

- è molto robusto su grande architetture con tantissimi nodi.
- ha la capacità di versionamento (gestisce la storia), è da attivare manualmente ma è una funzione nativa.
- capacità di compressione.

Svantaggi di HBase:

- non è facile gestirlo in progetti piccoli, non riesce a scalare verso il basso (si è costretti a mantenere non meno di 5 nodi nel cluster).
- non ha capacità di fare sorting e indicizzare gli oggetti al di là del Bloom Filter che può dire se una cosa è già stata inserita oppure no.
- i valori associati alle chiavi sono stringhe di bit, non viene associato alcun tipo di dato.

MongoDB

È un database di tipo “a documento”, questo oggetti di tipo testuale permette l’annidamento degli oggetti. Permette un linguaggio di interrogazione molto più ricco in grado di ricercare anche gli elementi annidati. L’aspetto positivo è che non si ha uno schema predefinito. I file XML e JSON ad esempio sono semi strutturati, non si è limitati a uno schema rigido.

Gli oggetti in MongoDB sono JSON, una struttura comporta da coppie key-value. Qualunque oggetto creato nel DB è a sua volta un oggetto Javascript.

Difficoltà: essendo lo schema dei JSON flessibile e semi-strutturato, può essere complicato effettuare le interrogazioni soprattutto nel caso di schemi molto grandi. Un esempio è quello dei nested objects e di \$elemMatch.

Le disgiunzioni con l’operatore \$or seguono la notazione polacca inversa

Nota:

- le parole che sono operatori all’interno delle interrogazioni si distinguono perché iniziano con \$.
- gli **update** si possono fare con senza \$set. Se non si usa \$set, l’oggetto viene sostituito comportando la perdita delle informazioni eventualmente presenti nel vecchio oggetto ma non introdotte nell’update. Con \$set è possibile aggiornare anche solo uno specifico campo dell’oggetto.
- findOne restituisce solo un oggetto (il primo) che fa match con la query
- in generale le query sono complicate
- gli indici vengono generati di default sugli identificatori
- Gli shortcut delle **decision functions** sono criticati perché soggetti a Query injection
- Attenzione: se si effettua una query di range (ad esempio sulla popolazione), la query fallisce se non tutti gli oggetti hanno quell’attributo (popolazione).

Replica sets and sharding: MongoDB mantiene più repliche dei dati mettendo più nodi in un organizzazione master-slave, questi nodi vengono detti **replica sets**. Di solito si consiglia di avere un numero dispari dei nodi qualora uno di questi fallisca (per esempio il master non è più raggiungibile), allora dovrebbe essere eletto un altro master e avendo un numero dispari di nodi si può leggere il master a maggioranza (quorum).

C’è la possibilità di fare **sharding**, ovvero partizionare una collezione di nodi della **replica set** per intervalli di valori (range di valori). Se alcuni di questi nodi non sono più raggiungibili, la parte di rete che rimane connessa con il nodo master rimane attiva e continua a rispondere sia alle operazioni di lettura che scrittura. In questo modo MongoDB si comporta sempre come disponibile in quanto cerca sempre di rispondere all’interrogazione (almeno per la porzione del master), i nodi che possono comunicare tra di loro eleggono un nuovo master con cui continueranno a lavorare.

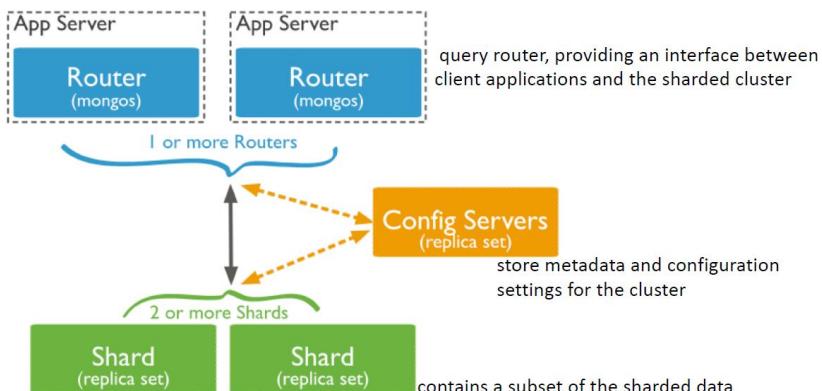
GridFS: File System distribuito che viene usato in MongoDB. Può fare query di tipo distribuito anche su dati di tipo spaziale mediante il pacchetto geoNear.

Replica sets and sharding in a shared cluster:

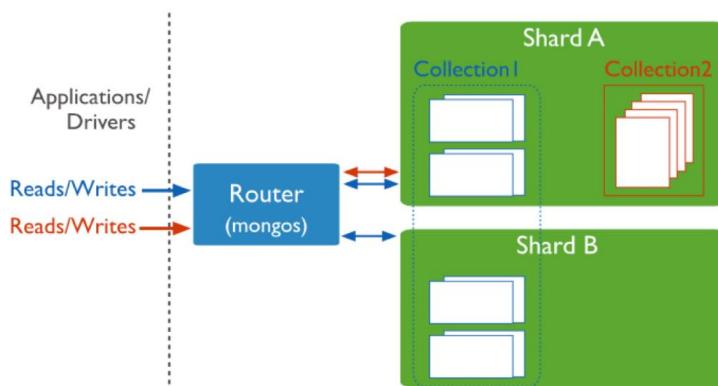
- elementi architetturali del sistema: diversi server applicativi su cui si esegue il processo **mongos** che instrada tutte le richieste provenienti dall’applicazione verso i replica set. Quindi **mongos** fa sempre

da interfaccia fra lo strato applicativo e quello di database. Il **config server** mantiene in memoria dei metà dati dei nodi dove sono allocate le diverse porzioni del dataset (ogni porzione del dataset viene detto **shard** che è allocato su un nodo diverso).

- Quello che nella nomenclatura di MongoDB è replica set, è il gruppo dei processi detti **mongod** che forniscono la ridondanza delle copie e l'alta disponibilità delle copie qualora avvenga una richiesta di un'operazione proveniente dallo strato applicativo. Tra tutte queste repliche ce n'è una primaria e gli altri secondari che si devono mantenere sincronizzati con quello primario. Per velocizzare le operazioni si sincronizzano e fare in modo che non ci sia un singolo point of failure, i secondari comunicano anche tra di loro e fanno questo grazie a un registro di log distribuito che si chiama **oplog** (un registro distribuito delle operazioni effettuate sulle repliche).



Applications drivers: dallo strato applicativo arrivano tutte le richieste di lettura e scrittura, queste passano attraverso lo strato di routing che instrada la richiesta al giusto shard dove quell'oggetto risiede. Ogni shard può contenere anche oggetti di collezioni diverse che sono stati allocati in quel nodo per rendere distribuita una collezione.

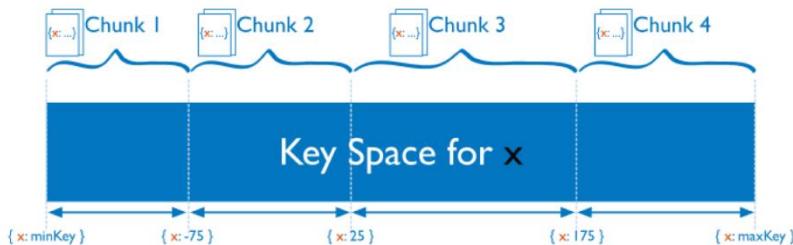


Partitioning documents into shards by key: queste collezioni si possono partizionare in base alla chiave **shard key** che potrebbe essere decisa

- in base alla cardinalità della chiave (quindi decidere quanti sono gli shard a seconda di quanto sono numerosi i valori distinti della chiave),
- per frequenza del numero di oggetti che hanno lo stesso valore della chiave oppure

- per range di cambiamento. Le chiavi molto dinamiche possono richiedere più lavoro e quindi si potrebbe fare in modo che queste potrebbero essere associate a chiavi di shard diverse, per esempio tramite l'uso di una funzione di hash.

La chiave di sharding verrà usata per partizionare la collezione su più nodi MongoDB. Le partizioni prendono il nome "Chunk 1, Chunk 2, ...". Questi vengono allocati nei vari nodi in base alla chiave di sharding.



È possibile utilizzare una funzione di hash per rimappare i valori di una chiave originaria in nuovi valori che distribuiscono gli oggetti della collezione attraverso i chunk. Bisogna fare in modo che i chunk siano il più possibile equilibrati in modo da non concentrare troppo traffico e lavoro in un solo nodo.

Chunk splitting: in modo trasparente al programmatore, ogni tanto MongoDB effettua il chunk splitting qualora di accorga che uno shard sia diventato troppo oneroso/grande, quando raggiunge una certa dimensione limite splitta in due i chunk. Avviene in maniera trasparente in base a una chiave per rendere bilanciati i due nuovi chunk.

Shards balancing: alternativa al chunk splitting, invece che spartire lo shard viene effettuato un bilanciamento dei chunk andando a migrare i chunk da uno shard (nodo) a un altro.

Nodo arbitro: può essere scelto per eleggere un nuovo nodo master nel caso in cui fosse necessario. Le repliche avvengono solo tra nodo primario e secondari, non con l'arbitro.

Neo4J

È un DB a grafo che

- è perfetto per dati completamente **non** strutturati, eterogenei e con molte proprietà eterogenee tra di loro,
- non ha tipi e non ha schema,
- non aggiunge vincoli di tipo particolare su come collegare i dati tra di loro,
- è molto veloce a navigare il grafo e seguire il pattern-matching,
- è in grado di supportare grafi giganteschi con miliardi di nodi e relazioni,
- ha molti strumenti aggiuntivi, ad esempio la ricerca sulle stringhe tramite gli indici dell'Apache Foundation,
- ha molti linguaggi e c'è anche un'interfaccia di tipo REST.
- Per molte applicazioni potrebbe essere l'unica struttura per navigare le informazioni.

Limiti:

- i grafi non sono riflessivi tramite lo stesso vertice,
- non gestisce lo sharding perché non è facile da far funzionare partizionare i grafi perché di solito sono molto connessi tra di loro,
- al momento della pubblicazione del libro non era un progetto completamente open source.

Note:

- Pensare al grafo in termini matematici: un insieme di nodi, che sono i vertici del grafo, e un insieme di archi che sono gli edges che permettono di passare da un vertice ad un altro del grafo.
- Questo modo di rappresentare i dati si focalizza in modo principale non tanto sulle proprietà dei nodi e degli archi quanto sulla navigazione del grafo. Si focalizza quindi maggiormente sulle relazioni tra i dati piuttosto che sulla somiglianza tra i valori. È possibile quindi istanziare diversi grafi con diverse etichette e quindi navigare il grafo seguendo specifici archi con specifiche etichette.
- Può memorizzare, in maniera naturale, dati altamente variabili (ogni attributo è opzionale).
- Modo di dire per i database relazionali: “on a long enough timeline, all fields become optional”, vuol dire che al passare di un tempo sufficientemente lungo, i requisiti sono cambiati talmente tanto che tutti i campi sono diventati opzionali.
- È molto **scalabile**: i tempi di navigazione sono il punto di forza di questo DB.
- Gestisce le repliche dei dati e prevede una tipologia master-slave nel cluster.
- **Complicazioni:** prevede le transazioni che godono delle proprietà ACID, infatti non è così banale partizionare il grafo in modo da fare sharding su più nodi e quindi gestisce il grafo alla vecchia maniera.
- **High Availability (HA):** HA permette di scrivere sui nodi slave che si sincronizzeranno con il nodo master che manterrà una copia e sincronizzerà gli slave. Permette quindi ai nodi slave di sincronizzarsi tra di loro e di scrivere prima ancora di aver comunicato con il nodo master (una scrittura in anticipo).

Groovy: è il linguaggio di programmazione che usa per fare delle query di tipo MapReduce sulla struttura e in aggiunta ha anche l'interfaccia di tipo REST. Può usare gli indici. Usa la libreria JUNG (Java Universal Network Graph) che è una libreria molto ricca di algoritmi molto sofisticati e avanzati di elaborazione e attraversamento di grafi.

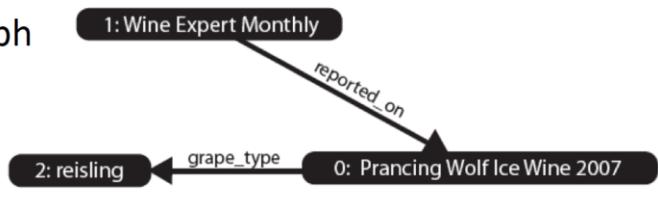
Gremlin: linguaggio di programmazione simile a jquery per interrogare il grafo. Is a query language for traversing graphs.

- può essere usato anche per navigare altri tipi di documenti come html
- nodi e archi hanno delle etichette che permettono di distinguergli
- g.V : restituisce i vertici/nodi
- g.E : restituisce gli archi

- Access all of the vertices of the graph

```
gremlin> g.V
==>v[0]
==>v[1]
==>v[2]
```

g represents the graph object



- Access all of the edges of the graph

```
gremlin> g.E
==> e[0][1-reported_on->0]
==> e[1][0-grape_type->2]
```

- List all properties of a vertex

```
gremlin> g.v(0).map()
==> name=Prancing Wolf Ice Wine 2007
```



- Filter a vertex by its name

```
gremlin> g.V.filter{it.name=='riesling'}
==> v[2]
```

- Find the outgoing edges of a vertex

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE
==> e[0][1-reported_on->0]
```



- Find the names of the vertexes connected to a given vertex by incoming edges

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE.inV.name
==> Prancing Wolf Ice Wine 2007
```

Cypher: è un altro linguaggio ma SQL-like, serve per restituire un insieme di elementi del grafo che soddisfano lo stesso pattern. È simile a SQL con la differenza che si implementa il pattern-matching tramite nodi e archi.

- It is a pattern matching language with a SQL-like syntax
- Example of a query in Cypher that starts from a node (number 0) and navigates edge of type 'grape_type' to return the set of nodes
- START ice_wine=node(0)
MATCH (ice_wine) -[:grape_type]-> () <[:grape_type]-(similar)
RETURN similar



Before sentiment analysis: NLP and pre-processing of texts

NLP: Natural Language Processing.

micro-blogs: testi molto brevi, ad esempio i tweet. Questo comporta ulteriori complicazioni nel processamento dei messaggi perché gli utenti sono portati a concentrarsi su pochi caratteri andando quindi anche a “storpiare” le parole per adeguarsi al canale utilizzato.

POS: Part Of Speech text. i POS tags dipendono dal sistema ma anche dalle lingue.

Corpus-based knowledge acquisition: concetto di tipo statistico per scegliere l'interpretazione più frequente di una frase, nemmeno questa è esente da errore. Rende più veloce le procedure si **parsing** e **POS tagging** perché esclude le alternative meno probabili a prescindere.

Spesso le interpretazioni delle frasi dipendono anche da chi parla.

Note:

- il task dell'analisi del sentimento e riconoscimento delle emozioni sono una nuova generazione di task di tipo classificazione, ovvero che vogliono associare un'etichetta di classe/categoria al testo che viene analizzato.

Annotazioni sul progetto

Unicode 32 per le emoji. <https://pypi.org/project/emoji/>

Cartella “ConScore”: contiene i termini con relativi score (polarizzazioni) a seconda delle emozioni.

MongoDB: db di tipo documentale dato che bisogna trattare dei documenti di testo.

Obiettivo: produrre una word cloud per ognuna delle 8 emozioni di base.

All’orale: c’è anche una domanda di teoria ad esempio

- come fare lo sharding
- come fare le query i db di tipo key-value

La discussione di gruppo avviene in maniera individuale.