Asserzioni

- Le asserzioni in SQL-2 sono vincoli non associati a un attributo o a una tabella specifica, ma direttamente allo schema.
- Permettono di esprimere tutti i vincoli di integrità definiti nella tabella.
- Le asserzioni includono vincoli di tupla, di tabella, su più tabelle e vincoli che richiedono una cardinalità minima su una tabella.
- Ogni asserzione ha un nome, che ne consente l'eliminazione dallo schema.
- La sintassi per la definizione di un'asserzione è: **CREATE ASSERTION NomeAsserzione CHECK(condizione)**.
- I vincoli di integrità possono avere due politiche di controllo:
    - Vincoli immediati: verificati immediatamente dopo ogni modifica del database (es. primary key, unique, not null, foreign key); se non soddisfatti, viene eseguito un rollback parziale.
    - Vincoli differiti: verificati solo al termine della transazione; se non soddisfatti, l'intera sequenza di operazioni viene annullata (rollback).
- È possibile impostare il comportamento di un vincolo (immediato o differito) con **SET CONSTRAINT NomeVincolo IMMEDIATE|DEFERRED**.

Viste

- Una vista è una tabella virtuale il cui contenuto è definito da query su altre tabelle (tabelle base) o viste non ricorsive nello stesso schema.
- È una relazione non persistente, cioè non contiene tuple fisiche ma solo la sua definizione.
- Gli attributi della vista devono corrispondere uno a uno alle colonne prodotte dalla query, altrimenti la vista li eredita direttamente dalla query.
- Le viste permettono di eseguire operazioni di modifica come le tabelle, a differenza delle query semplici.
- Le viste sono utilizzate per offrire diverse prospettive degli stessi dati, semplificare le interrogazioni e abilitare nuove interrogazioni.
- La sintassi per creare una vista è: **CREATE VIEW NomeVista [(Lista di Attributi)] AS selectSQL [WITH [CASCADED | LOCAL] CHECK OPTION]**.

Viste Aggiornabili

- Le modifiche apportate a una vista si propagano alla tabella base, in un modo che lo standard SQL richiede sia univoco.
- SQL92 permette l'aggiornamento solo per viste basate su singole tabelle senza funzioni aggregate; ogni tupla della vista corrisponde a una tupla della relazione di origine.

- L'opzione **WITH CHECK OPTION** è necessaria per gli aggiornamenti di una vista, garantendo che le tuple risultanti continuino ad appartenere alla vista (l'aggiornamento non deve violare i predicati di selezione).
- Per le viste derivate da altre viste, le opzioni **LOCAL** e **CASCADED** definiscono se il controllo viene fatto solo sulla vista corrente (local) o se deve propagarsi (cascaded), con **CASCADED** come default.
- Una **SELECT SQL** è aggiornabile se non include **DISTINCT**, funzioni aggregate, **JOIN** (usa solo una tabella), **SUBQUERY** nella **WHERE clause**, **GROUP BY** o **HAVING**.
- Senza la clausola **WITH CHECK OPTION**, un aggiornamento su una vista può far sì che le righe modificate non soddisfino più i criteri della vista, rendendo la vista vuota quando interrogata.
- Con la clausola **WITH CHECK OPTION**, l'aggiornamento è impedito se entra in conflitto con i predicati di selezione della vista, garantendo che le tuple aggiornate rimangano nella vista.
- La clausola **CHECK OPTION** è **CASCADED** per impostazione predefinita, ciò significa che la sua conformità viene controllata in tutti gli oggetti che fanno riferimento alla vista.

Cancellazione di Viste e Asserzioni

- Per eliminare una vista si usa **DROP VIEW NomeVista [RESTRICT|CASCADE]**.
    - **RESTRICT** (opzione predefinita) consente l'eliminazione solo se la vista non è usata in altre definizioni di tabelle o viste.
    - **CASCADE** elimina la vista e tutte le altre viste o tabelle che fanno riferimento ad essa.
- Per eliminare un'asserzione si usa **DROP ASSERTION NomeAsserzione [RESTRICT|CASCADE]**.
    - **RESTRICT** (opzione predefinita) consente l'eliminazione solo se l'asserzione non è più in uso.
    - **CASCADE** consente l'eliminazione dell'asserzione indipendentemente dal suo utilizzo.

Viste per la Risoluzione di Query SQL

- Le viste possono semplificare query complesse suddividendole in sottoquery gestibili.
- Ad esempio, per calcolare il numero medio di impiegati per dipartimento del Politecnico di Bari, si può creare una vista temporanea (**NumImpiegatiDip**) che calcola il numero di impiegati per dipartimento, e poi interrogare la vista per ottenere la media.

Basi di Dati Attive (Trigger)

- Una base di dati è "attiva" quando integra un sottosistema per definire e gestire "regole attive" che seguono il paradigma ECA (Evento-Condizione-Azione).
- Molti database relazionali, sia commerciali (Oracle, MS SQL Server) che open source (PostgreSQL, MySQL), sono considerati attivi perché supportano i trigger.
- Il paradigma ECA definisce una regola attiva che, quando un evento (E) accade su una tabella **target** e la condizione (C) è vera, esegue un'azione (A).
- L'Evento (E) è una primitiva DML (INSERT, UPDATE, DELETE).
- La Condizione (C) è un predicato booleano SQL basato sulla clausola WHERE.
- L'Azione (A) è una sequenza di istruzioni SQL arricchite con costrutti procedurali specifici del DBMS (es. PL/SQL in Oracle, PL/pgSQL in PostgreSQL).

Proprietà delle Basi di Dati Attive

- **Comportamento Reattivo**: Il DB può reagire autonomamente agli eventi, in particolare alle modifiche delle istanze.
- **Processore delle Regole (Rule Engine)**: Questo componente cattura gli eventi, esegue le regole attive e gestisce l'alternanza tra le transazioni utente e le regole di sistema.
- **Indipendenza della Conoscenza**: Le azioni che prima erano codificate nei programmi applicativi ora sono integrate nello schema tramite il DDL. Ciò significa che:
    - Le regole possono essere condivise tra le applicazioni.
    - Non è necessario replicare le regole nelle applicazioni.
    - Le modifiche alle regole non richiedono modifiche nelle applicazioni.

Utilità delle Regole Attive

- **Gestione Interna (DBMS)**:
    - Gestire vincoli di integrità predefiniti (es. politiche ON DELETE CASCADE).
    - Calcolare attributi derivati.
    - Gestire dati duplicati.
    - Gestire eccezioni sollevate dalla violazione di vincoli.
- **Gestione Esterna**:
    - Codificare regole aziendali complesse che non possono essere rappresentate direttamente nello schema tramite CHECK o ASSERTION.
    - Non esistono schemi fissi per la codifica, ogni problema applicativo richiede un approccio specifico.

Trigger: Sintassi e Fasi

- La sintassi dei trigger non è standardizzata in SQL-92 e varia a seconda del DBMS.
- La sintassi generale per la creazione di un trigger include:
    - **CREATE TRIGGER NomeTrigger**: definisce il nome del trigger.
    - **modalità evento {, evento}**: specifica quando il trigger si attiva (e.g., **AFTER INSERT**, **BEFORE UPDATE**).
    - **ON TabellaTarget**: specifica la tabella su cui il trigger opera.
    - **[referencing referenza]**: clausola opzionale per fare riferimento ai valori **NEW** e **OLD**.
    - **[granularità]**: specifica se il trigger è **FOR EACH ROW** o **FOR EACH STATEMENT**.
    - **[when (condizione)]**: condizione opzionale che deve essere vera per l'esecuzione dell'azione.
    - **StatementSQL**: il blocco di codice SQL da eseguire.
- Le fasi di un trigger sono: **Attivazione**, **Valutazione** ed **Esecuzione**.
- **Attivazione**:
    - La **modalità AFTER** valuta il trigger immediatamente dopo l'evento ed è comune quando la modifica di una riga consente il superamento dei vincoli di integrità referenziale. Viene attivato solo se i nuovi dati soddisfano la condizione. Utili per applicazioni di audit, calcolo di dati derivati e gestione di politiche di reazione a vincoli di integrità.
    - La **modalità BEFORE** valuta il trigger prima dell'evento. Usato per impostare valori di colonne inserite tramite trigger, accedere a valori "nuovi" e "vecchi" prima della verifica. Utili per verificare dati duplicati e chiavi, e gestire eccezioni impedendo l'esecuzione di eventi che attivano il trigger.
    - La **modalità DIFFERITA** valuta il trigger alla fine della transazione.
- **Esecuzione**: **StatementSQL** può contenere modifiche a tuple che attivano altri trigger in cascata.
- **Granularità**:
    - **ROW-LEVEL (FOR EACH ROW)**: Il trigger viene attivato, verificato ed eseguito per ogni tupla coinvolta nell'evento. Questo comportamento è orientato alle singole tuple e le variabili **NEW** e **OLD** sono disponibili.
    - **STATEMENT-LEVEL (FOR EACH STATEMENT)**: Il trigger viene attivato, verificato ed eseguito una sola volta per tutte le tuple target dell'operazione. Questo è il comportamento predefinito.
- Le clausole **REFERENCING** e **WHEN** sono opzionali e sono utilizzate solo con i trigger a livello di riga.

Granularità e Referenza del Trigger

- Durante la definizione di un trigger, è possibile fare riferimento ai valori "vecchi" e "nuovi" delle tuple coinvolte:
    - **NEW**: rappresenta la tupla dopo un'operazione di **INSERT** o **UPDATE**.
    - **OLD**: rappresenta la tupla prima di un'operazione di **DELETE** o **UPDATE**.
- Per accedere a un campo specifico di una tupla si usa la *dot notation* (es. **NEW.campo**).
- La clausola **REFERENCING** permette di rinominare queste variabili (es. **REFERENCING NEW AS NuovaVar, OLD AS VecchiaVar**).
- Le variabili **NEW** e **OLD** sono disponibili solo a livello di riga (**FOR EACH ROW**).
- I trigger **FOR EACH STATEMENT** sono usati se l'azione del trigger deve sempre essere eseguita sull'intera tabella.
- I trigger **FOR EACH ROW** sono usati se l'attivazione, la valutazione o l'esecuzione del trigger richiedono la conoscenza dello stato precedente o successivo dell'evento.
- L'ordine di esecuzione dei trigger è: **BEFORE Statement-level**, **BEFORE Row-level**, **AFTER Row-level**, **AFTER Statement-level**.

Statement SQL

- Lo **StatementSQL** è composto da una parte dichiarativa (opzionale) e una parte esecutiva.
- La sintassi varia a seconda del linguaggio procedurale utilizzato dal DBMS.
- La parte dichiarativa serve per dichiarare variabili usando tipi definiti da SQL-92.
- La parte esecutiva contiene istruzioni SQL (**SELECT**, **INSERT**, **UPDATE**, **DELETE**) arricchite con strutture procedurali come **IF**, **ELSE**, **FOR** e **WHILE**.

Esempi di Regole Aziendali con Trigger

- **Riduzione Stipendio**: Un trigger può essere configurato per ridurre automaticamente del 10% lo stipendio di tutti gli impiegati quando la media dei salari supera i 5000 euro. Questo trigger si attiva **AFTER INSERT, DELETE, UPDATE** sulla tabella **Impiegato**, operando a livello di statement. La condizione per la riduzione viene verificata all'interno del **StatementSQL** usando una subquery.
- **Integrità Referenziale (ON DELETE SET NULL)**: Un trigger può gestire la cancellazione di un dipartimento, impostando a **NULL** il campo **dipartimento** negli impiegati associati. Questo trigger si attiva **AFTER DELETE** sulla tabella **Dipartimento** e opera **FOR EACH ROW**. La clausola **WHEN** verifica l'esistenza di impiegati associati prima di eseguire l'aggiornamento.
- **Gestione Dati Derivati**: Per aggiornare la quantità totale ordinata (**qtTot**) nella tabella **Totale** ogni volta che viene inserito un nuovo ordine. Il trigger si attiva **AFTER**

INSERT sulla tabella **Ordine**, **FOR EACH ROW**, aggiornando **qtTot** aggiungendo la nuova quantità (**NEW.qt**) per il prodotto e il fornitore specifici.

- **Audit delle Valutazioni Libro**: Per memorizzare le modifiche delle valutazioni di un libro in una tabella di audit (**Giudizio**), solo se la nuova valutazione è inferiore alla precedente. Questo trigger si attiva **BEFORE UPDATE** sulla tabella **Libro**, **FOR EACH ROW**. La clausola **WHEN** verifica la riduzione del valore e, se vera, viene inserita una nuova riga nella tabella **Giudizio** con i valori **OLD** e **NEW** e il **timestamp**. L'esecuzione della transazione sulla tabella **BIBLIOTECA** dipende dal successo dell'esecuzione del trigger.

Le Transazioni

- Una transazione è un'unità di lavoro elementare, dotata di proprietà di correttezza, robustezza e isolamento.
- Ogni transazione è incapsulata tra un **BEGIN OF TRANSACTION (BOT)** e un **END OF TRANSACTION (EOT)**.
- All'interno di una transazione, viene eseguito solo uno tra **COMMIT WORK** (per rendere le modifiche permanenti) o **ROLL BACK WORK** (per annullare le modifiche).
- Un sistema transazionale consente di definire ed eseguire transazioni.

Proprietà ACID delle Transazioni

- **Atomicità**: Una transazione è indivisibile; o tutte le sue operazioni sono completate con successo o nessuna di esse. In caso di errore prima del **COMMIT**, si esegue **UNDO** (annullamento) per ripristinare lo stato precedente. In caso di errore dopo il **COMMIT**, si esegue **REDO** (ripristino) per garantire che lo stato del database rifletta gli effetti della transazione.
- **Consistenza**: Una transazione non deve mai violare i vincoli di integrità del database. Se il database è consistente all'inizio di una transazione, deve rimanere consistente alla fine. Non devono esserci contraddizioni tra i dati.
- **Isolamento**: Ogni transazione viene eseguita in modo indipendente dalle altre. Gli effetti di transazioni concorrenti devono essere equivalenti a quelli di un'esecuzione sequenziale delle stesse. Il fallimento di una transazione non deve influenzare le altre transazioni in esecuzione.
- **-Durabilità (Persistenza)**: Gli effetti di una transazione completata (**COMMITted**) devono persistere indefinitamente, anche in caso di malfunzionamenti del sistema. Per garantire questo, tutte le operazioni sono registrate in un **log file** persistente.

Metodi per l'ottenimento delle Proprietà ACID

- **Controllore dell'Affidabilità**: Garantisce l'atomicità e la durabilità delle transazioni, assicurando che le transazioni non siano lasciate incomplete e che i loro effetti siano permanenti dopo il **COMMIT**.

- **Controllore della Concorrenza**: Assicura l'isolamento, garantendo che le transazioni eseguite simultaneamente non interferiscano tra loro e che il risultato sia equivalente a un'esecuzione seriale.
- **Compilatore del Data Definition Language (DDL)**: Nel DBMS, il DDL garantisce la consistenza, verificando il rispetto dei vincoli di integrità definiti.

Controllo di Affidabilità

- Il controllore di affidabilità assicura che:
    - Le transazioni non siano incomplete.
    - Gli effetti di un **COMMIT** siano permanenti.
    - Lo stato del sistema possa essere ripristinato in caso di guasti.
- Realizza i comandi transazionali (**BEGIN TRANSACTION**, **COMMIT WORK**, **ROLLBACK WORK**).
- Implementa le primitive di ripristino (**Ripresa a caldo**, **Ripresa a freddo**).
- Si occupa della scrittura dei file di **log**.

Organizzazione del File di Log (Transaction Log)

- Il **transaction log** contiene informazioni ridondanti per ricostruire il database in seguito a guasti.
- Registra le operazioni del database in ordine cronologico e sequenziale.
- L'ultimo blocco inserito nel log è chiamato **top**.
- I **record di log** includono:
    - **Log di transazione**: registra le operazioni sul database (Begin, Commit, Abort, Update, Insert, Delete), con ID della transazione e ID dell'oggetto.
    - **Log di sistema**: registra le operazioni eseguite dal sistema, come **CHECKPOINT** (per registrare lo stato delle transazioni attive) e **DUMP** (copia di backup del DB).
- **UNDO**: operazione per annullare (**disfare**) un'azione su un oggetto O.
- **REDO**: operazione per rifare (**riproporre**) un'azione su un oggetto O.
- I log file sono idempotenti, ovvero l'applicazione ripetuta di **UNDO** o **REDO** sulla stessa azione produce lo stesso risultato.

Log di Sistema: Checkpoint

- L'operazione di **checkpoint** è eseguita periodicamente dal gestore di affidabilità per registrare lo stato corrente delle transazioni attive (CHECK(T1, T2, ..., TN)).
- Serve per velocizzare e semplificare le operazioni di ripristino dopo un guasto.
- I passaggi per un **checkpoint** includono: sospensione delle operazioni di scrittura, commit e abort; trasferimento delle pagine modificate (**COMMIT**) in memoria di massa; scrittura sincrona del **checkpoint record** nel log; ripresa delle transazioni sospese.

Log di Sistema: Dump

- Il **DUMP()** è una copia completa e consistente (backup) dell'intero database su supporti stabili.
- Viene eseguito solo quando il sistema è inattivo (e.g., fine settimana, notte).
- Dopo il **DUMP**, un record corrispondente viene scritto sincronicamente nel log.

Gestione dei Guasti

- **Guasto di sistema (soft failure)**:
    - Perdita di dati nella memoria centrale a causa di problemi software (bug, crash) o interruzioni hardware (cali di tensione).
    - Porta il database a uno stato inconsistente (perdita di dati dal buffer), ma i dati in memoria di massa rimangono validi.
- **Guasto di dispositivo (hard failure)**:
    - Perdita di dati sia in memoria di massa che in memoria centrale.
    - L'analisi del **log** (su memoria stabile) permette di ricostruire il database prima del riavvio del servizio.
- Si assume che il file di **log** sia sempre disponibile su memoria stabile; la sua perdita è un evento catastrofico e irrimediabile.

Modello Fail-Stop

- Il modello **fail-stop** descrive un sistema che, in caso di guasto (**Fail**), passa a uno stato di **Stop**, dal quale può recuperare tramite un processo di **Boot** e **Ripristino** per tornare al **Funzionamento Normale**. Un riavvio soddisfacente indica il **Fine Ripristino**.

Ripresa a Caldo

- La **Ripresa a caldo** avviene in seguito a un guasto di sistema e si articola in quattro fasi:
    1. **Checkpoint-search**: Si cerca l'ultimo **checkpoint** nel **log** partendo dall'ultimo blocco (**top**) e andando a ritroso.
    2. **Costruzione dei set**: Si creano due insiemi, **REDO-set** (transazioni che hanno superato il **commit** e devono essere rifatte) e **UNDO-set** (transazioni che non hanno superato il **commit** e devono essere annullate).
    3. **Applicazione delle azioni UNDO**: Si scorre il **log** all'indietro per annullare le azioni delle transazioni nel **UNDO-set**. Il log può essere analizzato anche prima del **checkpoint** se la transazione attiva più vecchia è iniziata prima del **checkpoint**.
    4. **Applicazione delle azioni REDO**: Si rifanno le azioni delle transazioni nel **REDO-set**.

Ripresa a Freddo

- La **Ripresa a freddo** avviene a seguito di un guasto di dispositivo e si articola in tre fasi:
    1. **Accesso al dump**: Si accede all'ultimo **dump** (backup) del database e si ricopia la parte di dati danneggiati.
    2. **Correzione dei dati**: Si ripercorre il log in avanti, applicando le operazioni effettuate ai dati danneggiati, ricostruendo lo stato dei dati all'ultimo valore precedente il guasto.
    3. **Esecuzione ripresa a caldo**: Dopo aver ricostruito i dati, si esegue una **ripresa a caldo** per completare il ripristino.

Controllo di Concorrenza: Architettura Generale

- L'architettura generale del controllo di concorrenza del **DBMS** è composta:
    - **Gestore delle transazioni**: gestisce i comandi **BEGIN**, **COMMIT**, **ABORT**.
    - **Gestore dei metodi di accesso**: gestisce le operazioni di **READ** e **WRITE** sul database.
    - **Gestore della concorrenza (Scheduler)**: si occupa di coordinare l'accesso ai dati, interagendo con una **Tabella dei lock**.
    - **Gestore della memoria secondaria**: gestisce le operazioni di **READ**, **WRITE** sulla memoria secondaria (il database fisico).

Controllo di Concorrenza

- Il **DBMS** non può eseguire transazioni una alla volta in modo seriale per motivi di efficienza.
- L'obiettivo è eseguire transazioni in modo **seriale equivalente**, mantenendo gli stessi effetti di un'esecuzione seriale.
- Transazioni corrette eseguite in concorrenza riducono il tempo di risposta.
- L'esecuzione in concorrenza può portare a diverse anomalie:
    - **Perdita di aggiornamento (lost update)**: Una transazione scrive un dato dopo che un'altra transazione lo ha letto, e quest'ultima lo sovrascrive.
    - **Lettura sporca (dirty read)**: Una transazione legge un dato modificato da un'altra transazione che poi esegue un **rollback**, annullando la modifica.
    - **Lettura inconsistente (unrepeatable read)**: Una transazione legge un dato più volte e ottiene valori diversi perché un'altra transazione ha modificato il dato tra le letture.
    - **Aggiornamento fantasma (phantom update)**: Un'anomalia che porta il database a uno stato inconsistente, dove si possono avere aggiornamenti che non rispettano i vincoli.
    - **Inserimento fantasma (phantom insert)**: Si verifica quando una transazione effettua operazioni su dati aggregati (media, somma) e un'altra transazione inserisce nuove righe, alterando i dati aggregati.

Livelli di Isolamento

- Sono previsti quattro livelli di isolamento per le transazioni:
    - **READ UNCOMMITTED**: Permette transazioni di sola lettura senza bloccare i dati, ma consente anomalie da transazioni concorrenti.
    - **READ COMMITTED**: Rilascia immediatamente i lock sui dati letti e ritarda quelli in scrittura, evitando le letture sporche (**dirty read**), ma non altre anomalie.
    - **REPEATABLE READ**: Blocca i dati sia in lettura che in scrittura, ma solo sulle tuple coinvolte, senza impedire l'inserimento fantasma.
    - **SERIALIZABLE**: Garantisce la serializzabilità effettiva del codice, bloccando gli accessi alle tabelle utilizzate e prevenendo tutte le anomalie.

Schedule

- Uno **Schedule** è una sequenza di operazioni di lettura/scrittura presentate da transazioni concorrenti, che seguono l'ordine di esecuzione sul database.
- Uno **Schedule Seriale** è quando le azioni di ciascuna transazione compaiono in sequenza, senza essere intervallate da istruzioni di altre transazioni.

- Uno **Schedule Serializzabile** produce lo stesso risultato di uno **schedule seriale**.

View-equivalenza

- Le operazioni **Ri(X)** leggono da **Wj(X)** se l'operazione di scrittura precede la lettura e non ci sono altre scritture tra di esse (**Wj(X) → Ri(X)**).
- **Wj(X)** è una **scrittura finale** se è l'ultima scrittura sull'oggetto X nello **schedule**.
- Due **schedule** sono **view-equivalenti** se hanno la stessa relazione "legge da" e le stesse scritture finali.
- Uno **schedule** è **view-serializzabile** se è **view-equivalente** a uno **schedule seriale**.
- Determinare se uno **schedule** è **view-serializzabile** è un problema **NP-hard**, quindi poco pratico.

Conflict-equivalenza

- Un **Conflitto** si verifica quando due azioni operano sullo stesso oggetto e almeno una di esse è una scrittura, generando **conflitti Lettura-Scrittura (rw - wr)** o **Scrittura-Scrittura (ww)**.
- Due **schedule** sono **conflict-equivalenti** se presentano le stesse operazioni e ogni coppia in conflitto ha lo stesso ordine in entrambi.
- Uno **schedule** è **conflict-serializzabile** se esiste uno **schedule seriale conflict-equivalente**.
- L'insieme degli **schedule conflict-serializzabili (CSR)** è strettamente incluso nell'insieme degli **schedule view-serializzabili (VSR)**.

Grafo dei conflitti

- Per verificare se uno **schedule** è **conflict-serializzabile (CSR)**, si può usare il **grafo dei conflitti**.
- Ogni nodo nel grafo rappresenta una transazione.
- Un arco viene tracciato da **Ti** a **Tj** se esiste un conflitto tra un'azione di **Ti** e un'azione di **Tj**, e l'azione di **Ti** precede l'azione di **Tj**.
- Uno **schedule** è **CSR** se il suo **grafo dei conflitti** è **aciclico**.
- Questo metodo ha una complessità lineare ma è computazionalmente costoso e non è utilizzabile in contesti distribuiti.

Primitive di Lock

- Le operazioni di lettura/scrittura sono protette da tre primitive di lock:
    - **r_lock**: precede ogni operazione di lettura e è seguita da un **unlock**. Permette **lock** condivisi sulla stessa risorsa.
    - **w_lock**: precede ogni operazione di scrittura e è seguita da un **unlock**. Permette un **lock** esclusivo sulla stessa risorsa.
    - **unlock**: rilascia il **lock** sulla risorsa.
- Una transazione è ben formata rispetto al **locking** se:
    - Una richiesta di **lock** concessa implica l'acquisizione della risorsa.
    - Una richiesta di **unlock** concessa implica il rilascio della risorsa.
    - Una richiesta di **lock** non concessa mette la transazione in stato di attesa.
    - Il **lock condiviso** può essere trasformato in **lock esclusivo** (**lock upgrade**).
- I **lock** concessi sono memorizzati nella **tabella dei lock**, che indica lo stato della risorsa (libero, r_locked, w_locked), e quali richieste sono permesse.

Locking a Due Fasi (2PL)

- Nel **Locking a due fasi (2PL)**, una transazione non può acquisire nuovi **lock** dopo averne rilasciato uno.
- Si compone di due fasi:
    - **Fase crescente**: La transazione acquisisce tutti i lock necessari per le risorse.
    - **Fase calante**: La transazione rilascia i lock acquisiti.
- Transazioni ben formate che seguono il protocollo **2PL** sono serializzabili rispetto alla **conflict-equivalenza**.
- Tuttavia, la classe **2PL** è strettamente inclusa in **CSR** e non previene le letture sporche.

Locking a Due Fasi Stretto (Strict 2PL)

- Nel **Strict 2PL**, una transazione può rilasciare i lock solo dopo aver completato correttamente le operazioni di **commit** o **abort** (alla fine della transazione).
- I **Lock di predicato** sono definiti in base a condizioni e impediscono l'accesso e la scrittura a dati che soddisfano tale predicato.
- Diverse livelli di isolamento possono essere realizzati nel **2PL**:
    - **READ UNCOMMITTED**: La transazione non richiede lock e non osserva i lock esclusivi posti da altre transazioni.
    - **READ COMMITTED**: Richiede lock in lettura ma li rilascia subito, evitando **dirty read** ma non altre anomalie.

- **REPEATABLE READ**: Applica il **Strict 2PL** solo a livello di tupla, evitando tutte le anomalie tranne **phantom insert**.
- **SERIALIZABLE**: Applica il **Strict 2PL** e i **lock di predicato**, evitando tutte le anomalie.

Controllo basato su Timestamp (Metodo TS)

- Ad ogni transazione viene assegnato un **timestamp (ts)** all'inizio, che definisce il suo ordine.
- Lo **schedule** è accettato solo se riflette l'ordinamento seriale basato sui **ts** di ciascuna transazione.
- **WTM(x)** (Write Timestamp) è il **ts** della transazione che ha eseguito l'ultima scrittura su x.
- **RTM(x)** (Read Timestamp) è il **ts** maggiore tra quelli delle transazioni che hanno letto x.
- Regole di accesso basate su **ts**:
  - **r_t(x)** (read): se **t < WTM(x)**, la transazione viene annullata. Altrimenti, **RTM(x) = max{RTM(X),t}**.
  - **w_t(x)** (write): se **t < WTM(x)** o **t < RTM(x)**, la transazione viene annullata. Altrimenti, **WTM(x) = t**.
- Questo metodo può comportare l'annullamento di molte transazioni, portando a varianti **Multiversione**.

2PL vs Metodo TS

- **2PL** e **Metodo TS** sono due approcci per il controllo della concorrenza.
- Nel **2PL**:
  - Le transazioni rifiutate sono messe in **Attesa**.
  - L'ordine è imposto dai **conflitti**.
  - L'esito dell'attesa è un **incremento del tempo di blocco**.
  - Il problema principale è il **Deadlock**.
- Nel **Metodo TS**:
  - Le transazioni rifiutate vengono (**Uccise e riavviate**).
  - L'ordine è imposto dal **timestamp (TS)**.
  - L'esito dell'attesa è dettato da specifiche **condizioni di attesa**.
  - Il riavvio è molto lento (**> tempo attesa 2PL**).
- Il **Blocco Critico (Deadlock)** si verifica quando due o più transazioni sono in attesa l'una dell'altra.

- La probabilità di conflitto aumenta linearmente con il numero di transazioni e quadraticamente con il numero medio di risorse richieste.

Deadlock (1/2)

- **Timeout**: Le transazioni attendono un tempo prefissato. Allo scadere del tempo, il **lock** è rifiutato e la transazione viene annullata. Facile da implementare, ma la definizione del **timeout** è cruciale: un **timeout** troppo elevato ritarda la rilevazione dei **deadlock**, mentre uno troppo basso può identificare falsi **deadlock**, annullando inutilmente le transazioni.
- **Prevenzione (deadlock prevention)**:
    - **Allocazione preliminare dei lock**: La transazione richiede tutti i **lock** per le risorse all'inizio; non sempre possibile, poiché non tutte le risorse possono essere conosciute in anticipo.
    - **Uso di timestamp**: Una transazione attende solo se esiste una precedenza tra i **timestamp**.
    - **Uccisione transazioni**: Si applicano politiche **preemptive** (uccidono la transazione che possiede la risorsa) o **non-preemptive** (uccidono la transazione che richiede la risorsa).
    - **Starvation**: Se una transazione viene ripetutamente uccisa, si può mantenere lo stesso **timestamp** anche dopo il riavvio per evitarla.

Deadlock (2/2)

- **Rilevamento (deadlock detection)**:
    - Non impone vincoli sul sistema.
    - Controlla le tabelle dei **lock** a intervalli predefiniti o tramite **timeout**.
    - Analisi del **grafo delle attese** tra transazioni (**WAIT-FOR**) per identificare cicli, indicando un **deadlock**.

Struttura DBMS

- Il **DBMS** è composto da diversi gestori interconnessi:
    - **Gestore delle interrogazioni**: Decide le strategie di accesso ai dati per rispondere alle query.
    - **Gestore dei metodi di accesso**: Esegue l'accesso fisico ai dati secondo la strategia definita dal gestore delle interrogazioni.
    - **Gestore del buffer**: Gestisce il trasferimento delle pagine dal database alla memoria centrale.

- **Gestore della memoria secondaria**: Controlla l'affidabilità e la concorrenza, gestendo i dati sulla memoria secondaria (disco).

Gestione del Buffer

- Il **Buffer** è un'area di memoria centrale preallocata al **DBMS** e condivisa tra le transazioni.
- È suddiviso in pagine, con dimensioni pari ai blocchi di I/O del sistema operativo.
- Si basa sul **principio di località dei dati**, suggerendo che i dati usati di recente hanno maggiori probabilità di essere riutilizzati.
- Le primitive di accesso alle pagine sono: **FIX**, **SET DIRTY**, **USE**, **UNFIX**, **FLUSH**, **FORCE**.
- **FIX**: Richiede l'accesso a una pagina e la carica nel **buffer**, restituendo un puntatore. La pagina è considerata allocata a una transazione attiva.
    - Funzionamento: cerca la pagina nel **buffer**; se non presente, cerca una pagina libera (se modificata, esegue **FLUSH** in memoria di massa) e la legge.
    - In assenza di pagine libere, una politica **Steal** seleziona una pagina "vittima" e la scarica; una politica **No Steal** sospende la transazione.
    - Incrementa un contatore di utilizzo della pagina.
- **SET DIRTY**: Indica che una pagina è stata modificata (imposta un bit di stato).
- **USE**: Accede alla pagina caricata in memoria.
- **UNFIX**: Indica al **buffer manager** che la pagina non è più in uso (decrementa il contatore).
- **FORCE**: Trasferisce una pagina in modo *sincrono* dal **buffer** alla memoria secondaria (richiesta dal gestore di affidabilità per evitare perdite di dati).
- **FLUSH**: Trasferisce pagine in modo *asincrono* e indipendente dalle transazioni attive (decisione del gestore del **buffer** per recupero spazio o ottimizzazione).

Gestore dei Metodi di Accesso

- Il gestore dei metodi di accesso trasforma un piano d'accesso (dall'ottimizzatore) in una sequenza di accessi alle pagine.
- Utilizza moduli software (**metodi d'accesso**) per accedere e manipolare i dati.
- Identifica i blocchi da caricare e li comunica al **buffer manager**.
- È in grado di individuare valori specifici all'interno di una pagina.
- Primitive fornite:
    - Accesso a tuple specifiche tramite chiave o **offset**.
    - Inserimento, aggiornamento e cancellazione di tuple.
    - Accesso a un campo specifico di una tupla (**tupla + offset + lunghezza campo**).

Struttura di una Pagina

- Le pagine del database hanno una struttura interna ben definita:
    - **Block Header/Trailer (BH/BT)**: Contiene informazioni di controllo utilizzate dal file system a livello di blocco.
    - **Page Header/Trailer (PH/PT)**: Contiene informazioni di controllo specifiche della struttura interna della pagina (es. ID oggetto, puntatori a pagine successive/precedenti).
    - **Dictionary**: Contiene puntatori ai singoli elementi (tuple) all'interno della pagina.
    - **Free Space**: Spazio disponibile nella pagina.
    - **Data**: Insieme delle tuple.
    - **Checksum**: Bit di parità per il controllo degli errori.

Strutture Sequenziali

- **Tuple organizzate in blocchi in memoria secondaria in modo sequenziale**:
    - **Entry-Sequenced (Seriale)**: Le tuple sono inserite in ordine di immissione. Semplice per gli inserimenti, ma inefficiente per le ricerche (scansione seriale). La cancellazione marca la tupla come eliminata; la modifica è locale se la dimensione non cambia, altrimenti comporta cancellazione e riscrittura in coda. Spesso usata con strutture di supporto (es. indici).
    - **Array**: La posizione nell'array dipende da un campo indice. Richiede tuple di dimensione fissa. Le primitive usano il valore dell'indice; poco utilizzato.
    - **Sequenziale ordinata**: Le tuple sono ordinate in base a un campo chiave. Efficiente per interrogazioni su intervalli e raggruppamenti. Difficile per inserimenti e cancellazioni (richiede riorganizzazioni periodiche).

Strutture con Accesso Calcolato (Hash)

- L'accesso associativo ai dati è basato su un campo chiave.
- Le tuple non sono necessariamente ordinate in memoria di massa.
- Si sfrutta l'accesso diretto tipico degli **array**.
- I blocchi sono allocati in modo contiguo per il file.
- Una funzione **Hash(fileID, key)** restituisce un **blockID** tra 0 e **B-1**.
- **Folding**: La chiave viene suddivisa in parti per ottenere un valore intero positivo.
- **Hashing**: Il valore intero è trasformato in un indirizzo di blocco.
- **Collisioni**: Possono verificarsi quando chiavi diverse mappano allo stesso **blockID**; in tal caso, i **record** si accumulano nello stesso blocco fino all'esaurimento.

- **Catene di overflow**: Blocchi aggiuntivi vengono allocati e collegati al precedente, rallentando la ricerca.

Strutture ad Albero (Indici)

- L'accesso associativo ai dati dipende da uno o più campi chiave.
- **Indici primari**: Contengono i dati stessi o fanno riferimento a un file ordinato in base allo stesso campo dell'indice.
- **Indici secondari**: Non contengono i dati, ma supportano le operazioni sui dati. Ogni elemento contiene un valore di chiave **k** e l'indirizzo del **record** associato.
- **Strutture ad albero dinamiche**: Sono efficienti anche per gli inserimenti e mirano a un tempo medio di accesso costante. Esempi includono **B-tree** e **B+ tree**.
- Ogni albero è composto da **nodi** collegati tramite **puntatori**:
    - **Nodo radice**.
    - **Nodi intermedi**.
    - **Nodi foglia**.
- Ogni nodo coincide con una pagina (o blocco) del file system.
- Gli **alberi bilanciati** garantiscono un percorso di lunghezza costante tra radice e foglie.

B-tree: Struttura

- I nodi intermedi (**non foglia**) di un **B-tree** sono composti da:
    - Valore di chiave **Ki**.
    - Un puntatore diretto ai blocchi referenziati da **Ki**.
    - Un puntatore al sotto-albero che fa riferimento ai blocchi con chiave **Ki < K < Ki+1**.
- Nei **B-tree** non c'è collegamento diretto tra le foglie.

B+ tree: Struttura

- I **nodi foglia** di un **B+ tree** memorizzano tutti i valori e sono collegati tra loro da **puntatori** nell'ordine stabilito dalla chiave.
- I **nodi intermedi** sono composti da una sequenza di coppie (puntatore, chiave).
    - **P0** punta al sotto-albero con chiavi **K < K1**.
    - Ogni **Pi** punta al sotto-albero con chiavi **Ki ≤ K < Ki+1**.
    - **PF** punta al sotto-albero con chiavi **K ≥ KF**.

B+ tree: Ricerca e Inserimento

- Ogni nodo di un **B+ tree** contiene **F** valori di chiave e **F+1** puntatori (**fan-out**).
- **F** dipende dalla dimensione della pagina e viene scelto per ridurre i livelli dell'albero.
- L'accesso a una tupla con chiave **V** segue un percorso specifico a seconda del valore di **V** rispetto alle chiavi nel nodo, fino a raggiungere un nodo foglia.
    - Se l'indice è **index-sequential**, le foglie contengono le tuple.
    - Se l'indice è **indiretto**, le foglie contengono puntatori alle tuple.
- **Inserimento**:
    - La nuova pagina viene inserita in uno slot libero nelle foglie.
    - Se non ci sono slot liberi, si esegue uno **split** sulla foglia.
    - Viene inserito un nuovo puntatore nel nodo di livello superiore (potrebbe richiedere ulteriori **split**).

B+ tree: Cancellazione

- La cancellazione avviene "in loco" marcando lo spazio allocato come "*invalido*".
- Se la cancellazione riguarda una chiave dell'albero:
    - Si recupera il valore della chiave successiva.
    - Si aggiorna la chiave nell'albero.
    - Tutti i valori nell'albero devono corrispondere al database (consigliato ma non strettamente necessario).
- Se due pagine contigue nelle foglie diventano libere:
    - **Merge**: L'informazione disponibile viene unita in un'unica pagina.
    - È necessario modificare i puntatori del livello superiore (eliminando un puntatore), il che può portare a ulteriori **merge**.

Architetture Distribuite

- **Parallelismo**: Utilizzato per ottimizzare le prestazioni di componenti/sistemi OLTP (Online Transaction Processing) e OLAP (Online Analytical Processing).
- **Replicazione dei dati**: Consente di creare copie di collezioni di dati, esportabili tra nodi di un sistema distribuito, per massimizzare la disponibilità e l'affidabilità dei dati.
- **Interazione tra sistemi e prodotti diversi**:
    - **Portabilità**: Permette di trasportare programmi tra ambienti diversi (influenzando il tempo di compilazione).

    - **Interoperabilità**: Permette a sistemi eterogenei di interagire (influenzando il tempo di esecuzione).
- **Importanza degli standard**: La portabilità dipende dagli standard dei linguaggi (es. SQL), mentre l'interoperabilità dipende dagli standard dei protocolli di accesso ai dati (es. ODBC, JDBC).

Paradigma Client-Server

- Nel paradigma **Client-Server**, il **Client** è un **Software Applicativo** su un **Personal Computer**, che esegue **Interrogazioni SQL** e invia le richieste.
- Il **Server** è un **DBMS** che supporta più applicazioni su un sistema dimensionato per il carico transazionale, e invia i risultati.
- **Interrogazioni compilate staticamente (compile and store)**: Sottomesse una sola volta, richiamate più volte tramite procedure o servizi remoti (cursori).
- **Interrogazioni con SQL dinamico (compile and go)**: L'interrogazione viene inviata come stringa di caratteri.
- **Interrogazioni parametriche**: Assegnazione di parametri per l'esecuzione di interrogazioni o procedure.
- L'architettura **Client-Server** prevede un **Server multi-threaded** (un processo per diverse transazioni) e un **Dispatcher** che distribuisce le richieste e gestisce le code di risposta.

Basi di Dati Distribuite

- **Omogenea**: Tutti i server utilizzano lo stesso **DBMS**.
- **Eterogenea**: I server utilizzano **DBMS** diversi.
- Una transazione può coinvolgere più server.
- La **gestione distribuita dei dati** offre flessibilità, modularità e resistenza ai guasti, ma aumenta la complessità strutturale.
- La **frammentazione dei dati** può essere:
    - **Orizzontale**: Ogni **frammento Ri** è un insieme di tuple con lo stesso schema, risultato di una selezione sulla relazione **R**.
    - **Verticale**: Ogni **frammento Ri** è un sottoinsieme dello schema della relazione **R**, risultato di una proiezione sulla relazione **R**.

Frammentazione Orizzontale: Esempio

- Il database **LIBRI** con attributi (ISBN, Titolo, Anno, Pagine, Genere) può essere frammentato orizzontalmente in base al genere.

- Ad esempio, un frammento può contenere tutti i libri di genere "Fantasy", un altro i libri di genere "Thriller", e un altro i libri di genere "Classici".

Frammentazione Verticale: Esempio

- Il database **LIBRI** può essere frammentato verticalmente in base agli attributi.
- Ad esempio, un frammento può contenere (ISBN, Titolo, Anno), mentre un altro (ISBN, Pagine, Genere).

Frammentazione e Allocazione

- **Correttezza della frammentazione**:
    - **Completezza**: Ogni dato della relazione originale **R** deve essere presente in almeno uno dei suoi frammenti **Ri**.
    - **Ricostruibilità**: La relazione originale **R** deve poter essere interamente ricostruita a partire dai suoi frammenti.
- Ogni frammento **Ri** viene implementato come un file fisico su un server specifico (**allocazione**).
- Lo **schema di allocazione** è il **mapping** tra frammenti (o relazioni) e i server che li memorizzano.
- **Mapping non ridondante**: Ogni frammento è allocato su un unico server.
- **Mapping ridondante**: Lo stesso frammento è allocato su più server.

Livelli di Trasparenza (1/2)

- **Trasparenza di frammentazione**: La query rimane identica sia per database frammentati che non frammentati. Non è necessario sapere se il database è distribuito o frammentato.
- **Trasparenza di allocazione**: Il programmatore conosce la struttura dei frammenti ma non ha bisogno di specificare la loro allocazione.

Livelli di Trasparenza (2/2)

- **Trasparenza di linguaggio**: Richiede la specificazione della struttura e dell'allocazione dei frammenti.
- **Assenza di trasparenza**: Si verifica con **DBMS** eterogenei, dove ogni **DBMS** ha una propria sintassi SQL e richiede la specificazione sia della struttura che dell'allocazione dei frammenti.

Classificazioni delle Transazioni

- **Richieste remote**: Transazioni di sola lettura verso un singolo **DBMS** remoto.
- **Transazioni remote**: Transazioni con comandi SQL generici verso un singolo **DBMS** remoto.
- **Transazioni distribuite**: Transazioni che coinvolgono più **DBMS**, ma ogni comando SQL fa riferimento a dati su un singolo **DBMS**.
- **Richieste distribuite**: Transazioni più complesse, dove ogni comando SQL può fare riferimento a dati distribuiti su qualsiasi **DBMS**.

Tecnologia delle Basi di Dati Distribuite

- La consistenza e la durabilità delle transazioni non dipendono dalla distribuzione dei dati, ma dalle proprietà locali del **DBMS**.
- **Ottimizzazione delle interrogazioni**:
    - Rilevante solo per le richieste distribuite.
    - Eseguita dal **DBMS** che genera la richiesta.
    - Decompone la richiesta in sotto-interrogazioni rivolte a più **DBMS**.
- **Controllo di concorrenza (isolamento)** e **Controllo di affidabilità (atomicità)** sono aspetti cruciali.
- La distribuzione dei dati influenza solo l'ottimizzazione delle interrogazioni distribuite.

Controllo di Concorrenza (Basi di Dati Distribuite)

- Una transazione **ti** può essere suddivisa in più sotto-transazioni **tij** (dove **j** è l'indice del nodo).
- La serializzabilità locale (a livello di singolo scheduler) non è sufficiente per garantire la serializzabilità globale.
- La **serializzabilità globale** si ottiene se esiste un unico **schedule seriale S** (che coinvolge tutte le transazioni del sistema) equivalente a tutti gli **schedule locali Si**.
- È più facile garantire la **serializzabilità globale** se gli **scheduler locali** usano il protocollo **2PL** o il **metodo dei timestamp**.
    - Se ogni nodo applica il **2PL** ed esegue il **commit** atomicamente quando tutte le risorse sono bloccate, gli **schedule** risultanti sono globalmente serializzabili.
    - Se a un insieme di sotto-transazioni distribuite viene assegnato un unico **timestamp**, gli **schedule** risultanti sono globalmente seriali in base all'ordinamento indotto dai **timestamp**.

Rilevazione Distribuita dei Deadlock

- Per la rilevazione distribuita dei **deadlock** si possono utilizzare i **timeout** oppure algoritmi specifici.
- L'algoritmo di rilevazione dei **deadlock** tiene conto di due scenari:
    - Due sotto-transazioni della stessa transazione sono in attesa su **DBMS** distinti (una attende la fine dell'altra).
    - Due sotto-transazioni di diverse transazioni sono in attesa sullo stesso **DBMS** (una blocca i dati a cui l'altra vuole accedere).

Algoritmo di Rilevazione Distribuito

- L'algoritmo viene attivato periodicamente e segue un ciclo:
    1. **Generazione delle nuove sequenze di attesa**: Ogni **DBMS** locale crea le proprie sequenze di attesa.
    2. **Invio sequenze ai DBMS remoti**: Le sequenze di attesa vengono inviate ad altri **DBMS** del sistema distribuito.
    3. **Ricezione delle sequenze di attesa da altri DBMS**: I **DBMS** ricevono le sequenze da altri nodi.
    4. **Ricostruzione locale del grafo d'attesa**: Ogni **DBMS** locale ricostruisce il grafo d'attesa combinando le proprie sequenze con quelle ricevute.
    5. **Analisi locale dei deadlock**: Il grafo d'attesa viene analizzato localmente per identificare eventuali **deadlock**.
    - Se i **deadlock** sono identificati, vengono risolti annullando (**abort**) una delle transazioni coinvolte.

Controllo di Affidabilità: Commit a Due Fasi (2PC)

- Il **Commit a Due Fasi (2PC)** garantisce l'atomicità delle transazioni distribuite, assicurando che tutti i nodi coinvolti raggiungano lo stesso risultato (**commit** o **abort**).
- È un protocollo robusto ai guasti, dove il **Transaction Manager (TM)** e i **Resource Manager (RM)** scrivono record nei loro **log file**.
- Il **TM** scrive i record di **prepare** (con l'identità dei **RM**), **global commit** o **global abort** (che decide l'esito finale della transazione su tutti i nodi), e **complete** (alla fine del protocollo).
- Ogni **RM** rappresenta una sotto-transazione e scrive i record di **begin**, **insert**, **delete**, **update** (come in un sistema centralizzato), e **ready** (quando è irrevocabilmente pronto a partecipare al **commit**, includendo l'identificatore del **TM**).

Protocollo in Assenza di Guasti (1/2) (2PC)

- Il **TM** può comunicare con i **RM** tramite **broadcast** o comunicazione seriale.
- **Prima Fase**:
    1. Il **TM** scrive il record di **prepare** e invia un messaggio di **prepare** per avviare il protocollo. Un **timeout** predefinito indica il ritardo massimo per ricevere risposte dai **RM**.
    2. I **RM** (se in stato affidabile) ricevono il messaggio, scrivono il record di **ready** (se pronti a partecipare) e inviano il messaggio di **ready** al **TM** (o **not-ready** in caso di guasto della transazione).
    3. Il **TM** raccoglie le risposte: se tutte sono positive, scrive **global commit** nel **log**; altrimenti, scrive **global abort**.

Protocollo in Assenza di Guasti (2/2) (2PC)

- **Seconda Fase**:
    1. Il **TM** invia la sua decisione **globale** ai **RM** e imposta un **timeout** per l'attesa dei messaggi di risposta.
    2. I **RM**, in stato di **ready**, ricevono il messaggio e scrivono il record di **commit** o **abort** (questa volta **locale**).
    3. I **RM** inviano un messaggio di **acknowledgement (ack)** al **TM**.
    4. L'implementazione del **commit/abort** procede su ciascun server.
    5. Il **TM** raccoglie i messaggi di **ack**: se tutti arrivano, scrive un record di **complete**; altrimenti, reimposta un nuovo **timeout** e ripete la trasmissione finché tutti i **RM** non hanno risposto.

Ottimizzazioni del 2PC

- Il **2PC** è un protocollo costoso a causa delle scritture sincrone nel **log** (tramite **FORCE**) per garantire la persistenza.
- **Esito di default in assenza di informazione**:
    - **Protocollo di abort presunto**: Se non ci sono informazioni sufficienti, si presume un **abort**. Ogni richiesta di recupero remoto (**remote recovery**) porta a un **abort**. Solo i record di **ready**, **commit** (**RM**) e **global commit** (**TM**) devono essere scritti in modo sincrono (altri non sono critici per il recovery).
    - **Protocollo di commit presunto**: Una transazione di sola lettura in caso di operations di sola lettura non deve influenzare l'esito finale della transazione.
- **Sola lettura**: Se le operazioni sono di sola lettura, il **RM** non deve influenzare l'esito finale della transazione. Al messaggio di **prepare**, i partecipanti "sola lettura" avvisano il **TM** con un messaggio **read-only**, che viene ignorato nella seconda fase del protocollo.

Analisi dei Dati: OLAP, Data Warehousing, Data Mining

- La tecnologia delle basi di dati **OLTP (OnLine Transaction Processing)** gestisce principalmente i dati in linea.
- L'analisi dei dati passati e correnti (**OnLine Analytical Processing - OLAP**) è utile per la pianificazione e la programmazione delle attività future.
- Un **Data Warehouse (DW)** è un deposito di dati, contenente dati opportunamente analizzati per supportare le decisioni.
- I sistemi **OLTP** fungono da *data source* per l'ambiente **OLAP**.
- Gli *utenti* si differenziano: i **terminalisti** operano in **OLTP**, mentre gli **analisti** operano in **OLAP**.

Sistemi OLTP e OLAP: Confronto

- **OLTP (OnLine Transaction Processing)**:
    - Finalità: Gestione dei dati.
    - Operazioni: Set ben definito.
    - Dati: Quantità limitata, bassa complessità.
    - Sorgenti Dati: DB singolo.
    - Variabilità: Continuo aggiornamento, stato del sistema in tempo reale.
    - Proprietà ACID: Rispettate.
- **OLAP (OnLine Analytical Processing)**:
    - Finalità: Analisi dei dati.
    - Operazioni: Non previste nella progettazione del DB (sistemi di supporto decisionale).
    - Dati: Grosse moli di dati.
    - Sorgenti Dati: DB eterogenei e distribuiti.
    - Variabilità: Dati storici aggiornati a intervalli regolari.
    - Proprietà ACID: Non rilevanti, operazioni di sola lettura.

Caratteristiche dei Data Warehouse

- Utilizzano dati provenienti da più **DB** eterogenei.
- I meccanismi di importazione sono **asincroni** e **periodici**, per non penalizzare le prestazioni delle *data source*.
- Il **Data Warehouse** non contiene dati perfettamente allineati con il flusso di transazioni degli **OLTP**.

- La **qualità dei dati** è cruciale: la semplice raccolta non è sufficiente per un'analisi corretta; i dati possono contenere inesattezze, errori o omissioni.

Architettura di un Data Warehouse (DW)

- L'Architettura del Data Warehouse include diversi componenti:
    - **Data Source**: Le fonti di dati possono essere di qualsiasi tipo, inclusi **DBMS** tradizionali, **legacy system** o dati non gestiti da **DBMS**.
    - **Data Filter**: Controlla la correttezza dei dati prima dell'inserimento nel warehouse, eliminando dati scorretti e correggendo incoerenze tra diverse fonti. Essenzialmente, pulizia dei dati (**data cleaning**) per garantire un'elevata qualità.
    - **Export**: L'esportazione dei dati è incrementale; il sistema raccoglie solo le modifiche (inserzioni o cancellazioni) dalle **data source**.
    - **Acquisizione dei Dati (loader)**: Responsabile del carico iniziale dei dati nel **DW**. Predispone i dati per l'uso, ordina, aggrega e costruisce le strutture dati del **warehouse**. Le operazioni di acquisizione avvengono in **batch** quando il **DW** non è in uso. L'allineamento dei dati avviene in modo incrementale.
    - **Allineamento dei Dati (refresh)**: Propaga incrementalmente le modifiche dalle **data source** al **DW**. Può avvenire tramite *data shipping* (invio degli aggiornamenti a archivi variazionali, con trigger nelle data source che registrano le modifiche) o *transaction shipping* (che usa il log delle transazioni per costruire gli archivi variazionali).
    - **Accesso ai Dati**: Modulo chiave per l'analisi dei dati. Realizza interrogazioni complesse (join, ordinamenti, aggregazioni) e supporta nuove operazioni come **roll up**, **drill down** e **data cube**.
    - **Data Mining**: Utilizza tecniche algoritmiche per dedurre informazioni "nascoste" dai dati, svolgendo ricerche sofisticate e esplicitando relazioni implicite.
    - **Export dei dati**: Consente l'esportazione dei dati da un DW ad un altro, supportando architetture gerarchiche.
- **Moduli di ausilio**: includono un componente per sviluppare il DW (che facilita la definizione dello schema e i meccanismi di importazione) e un dizionario dei dati (glossario) che descrive il contenuto del DW per facilitare l'analisi.

Schema di un Data Warehouse

- Nella costruzione di un **DW** aziendale, si focalizza su sottoinsiemi semplici di dati dipartimentali (es. **data mart**).
- Ogni **data mart** è organizzato secondo uno **schema multidimensionale** o **schema a stella**.

Schema a Stella: Caratteristiche

- Il **data mart** segue uno **schema a stella** con una **unità centrale** che rappresenta i **fatti** (le misure) e diverse **unità a raggiera** che rappresentano le **dimensioni** dell'analisi.
- Le **relazioni sono uno a molti**: ogni occorrenza di fatto è collegata a una e una sola occorrenza di ciascuna dimensione.
- Il **fatto** ha una chiave composta dagli attributi chiave delle dimensioni; gli altri attributi sono misure (**numerici**).
- La struttura è regolare e indipendente dal problema.
- Sono necessarie almeno due dimensioni, altrimenti lo schema degenera in una semplice gerarchia uno-molti.
- Un numero elevato di dimensioni è sconsigliato, poiché complica la gestione dei fatti e l'analisi.
- I dati derivati e le ridondanze sono presenti nella dimensione del tempo per facilitare l'analisi.
- I fatti sono in **forma normale di Boyce-Codd** (ogni attributo non chiave dipende funzionalmente dalla sua unica chiave).
- Le dimensioni sono generalmente **relazioni non normalizzate**, per evitare operazioni di **join**.

Schema a Fiocco di Neve

- Lo **schema a fiocco di neve** è un'evoluzione dello **schema a stella** che struttura gerarchicamente le dimensioni, normalizzandole.
- Questo schema rappresenta esplicitamente le gerarchie, riducendo ridondanze e anomalie.
- Le dimensioni, che nello **schema a stella** sono spesso non normalizzate, qui sono decomposte in tabelle più piccole e normalizzate, legate tra loro.

Analisi dei Dati: Operazioni

- Le operazioni standard per formulare **query** nell'analisi dei dati includono:
    - **Drill down**: Permette di aggiungere nuove dimensioni di analisi, disaggregando i dati.
    - **Roll up**: Permette di eliminare dimensioni di analisi, aggregandole.
    - **Data Cube (Slice-and-dice)**: Seleziona un sottoinsieme delle celle di un cubo (affettamento e taglio) e proietta, riducendo la dimensionalità dei dati.
- L'analisi dei dati in un **data mart** organizzato a stella richiede l'estrazione di un sottoinsieme di fatti e dimensioni.

- Le dimensioni sono usate per selezionare e raggruppare i dati.
- I fatti sono normalmente aggregati.
- Possono essere creati moduli predefiniti per estrarre i dati con selezioni, aggregazioni e valutazioni di funzioni aggregate.
- L'operazione di **roll up** può essere eseguita direttamente sui risultati di un'interrogazione.
- L'operazione di **drill down** richiede una riformulazione dell'interrogazione, poiché coinvolge dati non presenti nell'interrogazione iniziale.

Esempi di Drill-down e Roll-up

- **Drill-down**: Un manager interessato alle vendite di pasta per zona (Nord, Centro, Sud) nel trimestre Feb-Apr, partendo da un totale per mese, può eseguire un **drill-down** sulla dimensione "Zona" per visualizzare le vendite disaggregate per ogni zona.
- **Roll-up**: Un manager interessato alla suddivisione delle vendite di pasta per zona (Nord, Centro, Sud) per l'intero trimestre, può eseguire un **roll-up** sulla dimensione "Mese" per aggregare le vendite mensili.

Rappresentazione Multidimensionale (Cube)

- I dati in un **Data Warehouse** sono spesso rappresentati in un **cubo multidimensionale**, che permette di visualizzare i dati lungo diverse dimensioni (es. Tempo, Prodotto, Regione).
- Il **cubo** contiene le **misure** (i fatti) e le **dimensioni** che definiscono le prospettive di analisi.

Slice-and-dice

- **Slice-and-dice** è un'operazione che seleziona un sottoinsieme delle celle di un **data cube**, ottenendo cubetti del **cubo** stesso. Questa operazione permette di esplorare i dati riducendo la dimensionalità.
- Ad esempio, un manager può voler analizzare le vendite di un prodotto specifico in un certo periodo e in tutti i mercati, o le vendite totali in tutti i mercati per un periodo specifico. Questo si realizza "affettando" il cubo lungo le dimensioni desiderate.

Data Mining

- Il **Data Mining** si occupa della ricerca di informazioni "nascoste" all'interno dei **Data Warehouse**.
- Esempi di utilizzo includono:
    - **Analisi di mercato**: Identificare prodotti acquistati insieme o in sequenza.

- **Analisi di comportamento**: Rilevare frodi o usi illeciti (es. carte di credito).
- **Analisi di previsione**: Stimare costi futuri (es. cure mediche).

Fasi del Processo di Data Mining

- Il processo di **data mining** si articola in cinque fasi:
    1. **Comprensione del dominio**: Acquisizione di conoscenza sul contesto dei dati.
    2. **Preparazione sul set di dati**: Individuazione di un sottoinsieme di dati dal **DW** e loro codifica per l'algoritmo.
    3. **Scoperta dei pattern**: Ricerca e individuazione di schemi ripetitivi nei dati.
    4. **Valutazione dei pattern**: Valutare gli *esperimenti* da condurre, le *ipotesi* da formulare e le *conseguenze* da trarre dai pattern scoperti.
    5. **Utilizzo dei risultati**: Prendere decisioni operative basate sul processo di **data mining** (es. allocazione merci, concessione credito).

Regole di Associazione

- Le **regole di associazione** mirano a scoprire relazioni di tipo causa-effetto tra i dati.
- Un esempio è la **Basket Analysis**, che analizza dati raggruppati per transazioni di acquisto.
- Una **regola associativa** è composta da una **premessa** e una **conseguenza** (es. "Pannolini → Birra").
- Si possono definire metriche per le probabilità associate alle regole:
    - **Supporto**: Probabilità che sia la **premessa** che la **conseguenza** siano presenti in un'osservazione.
    - **Confidenza**: Probabilità che la **conseguenza** sia presente in un'osservazione, data la presenza della **premessa**.
- Il problema del **data mining** è quello di trovare tutte le **regole di associazione** con supporto e confidenza superiori a valori prefissati.

Discretizzazione

- La **discretizzazione** consente di rappresentare un intervallo continuo di dati utilizzando un numero limitato di valori discreti.
- Permette di rendere più evidente il fenomeno oggetto di osservazione.
- Ad esempio, gli stipendi continui possono essere discretizzati in "Basso" (< 1000), "Medio" ($1000 \leq$ stipendio < 2500), e "Alto" (stipendio ≥ 2500).

Classificazione

- La **classificazione** consiste nel catalogare un fenomeno in una classe predefinita utilizzando algoritmi di classificazione (es. alberi decisionali).
- Quando i fenomeni sono descritti da un gran numero di attributi, i classificatori determinano gli attributi significativi, separandoli da quelli irrilevanti.
- Un esempio è la classificazione del salario in base all'età e allo stipendio, distinguendo tra impiegati "Di linea" o "Low Cost".