- DB relazionali
  - Design logico: informazioni suddivise in tuple
  - Dati persistenti
  - Query standard
  - Integrazione: database condiviso tra più applicazioni
  - Non funzionano bene sui cluster
  - Implementano le relazioni tramite chiavi esterne (foreign key)
  - join costose in caso di relazioni complesse.

- Impedance Mismatch
  - Differenza tra modello relazionale e le strutture dati rappresentate in memoria
  - Necessaria una traduzione delle strutture di dati in memoria (applicazione) in tuple adatte al DB relazionale

- Object DB
  - Replicano la struttura degli oggetti in memoria sul disco "risolvendo" l'impedance Mismatch
  - Può ignorare troppo il database compromettendo le prestazioni delle query

- Application DB
  - Struttura del DB conosciute solo da chi implementa il codice dell'applicazione
  - Disaccoppia il database dai servizi ottenendo maggiore libertà di scelta del modello di DB
  - Il modello scelto è spesso il RDB per semplicità e familiarità

- Aggregati
  - I modelli ad Aggregati funzionano bene sui cluster (un aggregato su uno stesso cluster)
  - Un aggregato è una collezione di oggetti correlati (cliente, ordine) trattati come una unità
  - Il design degli aggregati dipende fortemente dal contesto

- ACID
  - Atomicity: la transizione viene completata o non effettuata affatto
  - Consistency: si salvano solo dati validi
  - Isolation: le transizioni non hanno effetto le une con le altre
  - Durability: le scritture non devono essere perse
  - ACID vale per singoli aggregati, per aggregati multipli la gestione si lascia alle applicazioni
  - I DB relazionali offrono la proprietà acid per operazioni su più righe

- Key-Value Database
  - Coppie <chiave,valore> dove l'aggregato è un blob per il DB
  - Si può accedere solo per chiave (questo è la limitazione principale)
  - Possiamo applicare le reference (ad esempio inserire orderID in customerID) per ottimizzare le letture sfruttando gli aggregati
  - In generale le chiavi devono avere una struttura logica (per estensibilità e leggibilità) e un significato (avere una chiave tipo *123e45tp67*, diventa inutile). Esiste infatti una convenzione che si basa sulle seguenti regole:
    - Le componenti devono essere intuitive. Ad esempio, possiamo utilizzare "cust" per indicare "customer"

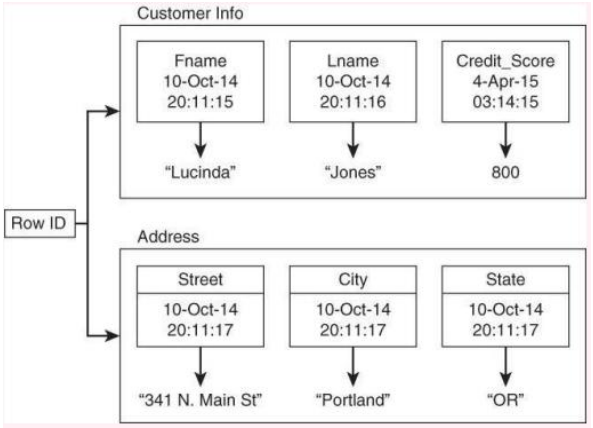
        - Usare componenti range-based. Come, ad esempio, *cust<date>:<purhcase>:custID*, ci permette di prelevare tutti i clienti che hanno fatto un acquisto in una determinata data.
        - Utilizzare un delimitatore in modo consistente. Ad esempio ":"
        - Le chiavi devono essere brevi, senza sacrificarne la comprensibilità.
    - In genere è possibile partizionare il database, assegnando le coppie key-value a nodi diversi in un cluster, utilizzando l'Hashing.
    - Non supportano le range query (ad esempio ricerca di clienti tra i 18 e 25 anni)
    - Più grandi sono i valori, più costa eseguire ricerche su parti di essi. La scelta della dimensione degli aggregati dipende dalle operazioni che dobbiamo eseguire.
    - Design patter per Key-Value DB:
        - Time To Live (**TTL**) **Keys**. Risulta utile nel caso in cui i dati abbiano una scadenza naturale, come ad esempio per l'acquisto di un biglietto. La chiave rimane finché l'utente non finalizza il pagamento, in caso contrario la chiave scade. Durante la durata della chiave, il biglietto corrispondente è bloccato.
        - **Emulating tables**. Non risulta pratico emulare tabelle relazionali, ma se il numero di tabelle emulate sono poche allora possiamo salvare le informazioni di un aggregato come se fosse una tabella.
          Ad esempio:
          ```
          define addCustRecord (p_id, p_name, p_addr, p_city)
          begin
              setCustAttr(p_id,'name', p_name);
              setCustAttr(p_id,'addr',p_addr);
              setCustAttr(p_id,'city',p_city);
          end;
          ```
        - **Atomic Aggregates**. Questo tipo di aggregati contiene tutti i valori che devono essere aggiornati insieme o altrimenti non aggiornati affatto. Se compriamo un biglietto, tutte le informazioni al riguardo devono essere aggiornate (la data, il luogo e il posto assegnato).
        - **Enumerable Keys**. Per generare nuove chiavi vengono utilizzati dei Counter o delle Sequenze. Oppure utilizzare la data (es.: *tcktLog:20240327:1*, *tcktLog:20240327:2*)

- Document Database
    - Ogni documento (aggregato) ha una chiave associata
    - Coppie <chiave,documento> con struttura visibile al DB
    - Si può accedere per chiave o query basate sul documento
    - A differenza del key-value DB dove ogni attributo è in una coppia separata, nei Document DB gli attributi sono archiviati in un singolo oggetto (facilita la ricerca tipo "*last_purchase > date*" che in key-value ne avremmo dovute fare n: *cust_list, cust_name* etc...)
    - Le collezioni sono liste di documenti; i documenti nella stessa collezione dovrebbero avere una struttura comune (mentre non necessario per diverse collezioni), ma possono benissimo aggiungere attributi arbitrari e ciò migliora la flessibilità (es.: solo 2% diversi).
    - Le collezioni contengono documenti che rappresentano istanze di un'entità. In generale si cerca di evitare entità troppo astratte, risultando in una bassa coesione nella collezione, poiché potrebbe risultare inefficiente nei filtering. In alcuni casi è infatti preferibile separare la singola collezione in più collezioni piuttosto che lasciare una collezione con diversi attributi.
    - In alcuni casi è necessario applicare diversi join per ottenere un determinato risultato, diventando troppo complesso. In questi casi viene utilizzata la **denormalizzazione**, ossia

salvare i dati che vengono usati più frequentemente insieme nella stessa struttura (ad esempio le informazioni dei giocatori, all'interno di una squadra).

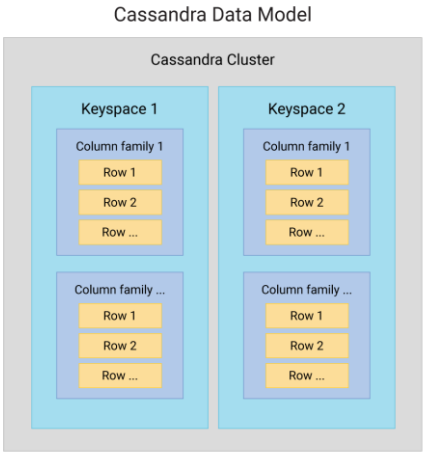
- One-to-Many: L'entità unica (One) è il documento primario, mentre i many sono salvati all'interno. Come la squadra (one) e i giocatori (many)
- Many-to-Many: Ogni collezione ha una reference a delle istanze dell'entità. Ad esempio, se abbiamo un documento in cui salviamo le informazioni di alcuni prodotti in delle categorie, allora il documento delle categorie avrà un richiamo all'interno degli id dei prodotti presenti in quella categoria (parent reference). Allo stesso modo i singoli prodotti avranno un richiamo con l'id della categoria a cui appartengono (Child references)

- Column-Family Database
  - Gruppi di colonne (famiglia) archiviate insieme (poiché spesso usati insieme)
  - Facilita la lettura (la scrittura nei DB avviene per righe, mentre di solito c'è il bisogno di leggere più valori della stessa colonna)
  - Architettura a due livelli:
    - Coppia <id riga, aggregato>
    - L'aggregato è costituito da valori (colonne) del tipo <col key, col val>



  - Esempio di accesso: get(id_riga, col_val)
  - Il vantaggio è che le colonne sono ordinate, permettendoci di nominare le colonne più frequentemente utilizzate in modo che vengano recuperate per prime.
  - Per modellare i dati, è importante ricordare di farlo in base ai requisiti di query e non per il fine della scrittura; la regola generale è rendere facile la query e denormalizzare i dati durante la scrittura.
  - Le colonne di una stessa famiglia sono immagazzinate insieme e le operazioni write/read sono atomiche.

- CASSANDRA DB (column family DB)
  - Wide-column DB: le colonne possono essere aggiunte a piacimento senza bisogno di chiavi esterne.
  - Colonna:
    - Ogni colonna ha un nome/chiave e un valore
    - Ogni colonna ha un timestamp usato per risolvere conflitti di replica
    - Potrebbe avere un TTL (data di scadenza della colonna)



- Riga:
    - Collezione ordinata di colonne identificata da una chiave
    - I dati di una riga dovrebbero entrare in un singolo nodo (del cluster)
    - Le colonne di una riga sono spesso sparse ed ogni riga ne contiene un numero arbitrario
- Tabelle:
    - Una tabella è un container di righe (sinonimo di famiglia di colonne)
    - Una chiave primaria unica definisce una riga
    - Le righe possono essere aggiunte a runtime senza restrizioni
    - Le tabelle non sono connesse da valori (no chiavi esterne) e possono essere distribuite su nodi multipli
- KeySpace
    - È una istanza di un Cassandra database ed è un container di tabelle
- Storage
    - Cassandra usa un'architettura peer to peer dove i nodi possono essere aggiunti o rimossi da un cluster
    - Ogni nodo condivide dati con altri nodi, da uno a tre (gossip protocol)
    - I conflitti tra nodi diversi vengono risolti confrontando due Markle trees (famiglie di colonne rappresentate come alberi hash binari dove le foglie sono i singoli valori di una chiave) (uno dal nodo mittente, uno dal destinatario). Se non corrispondono il mittente determina chi ha i dati più recenti e aggiorna il destinatario con i dati più recenti.
    - Hinted handoff: Se un nodo vuole fare una scrittura su un altro nodo che al momento non è disponibile. La scrittura è salvata in un nodo proxy che aspetta finché il nodo non diventa disponibile.

- Graph Database
    - Permette di catturare relazioni complesse
    - Modelli di dati naturalmente visti come grafi (orientati) dove i nodi sono gli attributi e gli archi le relazioni (come FRIEND_OF)
    - Le query navigano sul grafo attraverso nodi e archi (necessario uno starter point)
    - Eseguito di norma su singoli server ed ha query basate sui grafi
    - Le transizioni mantengono la consistenza per nodi e archi multipli
    - Non funziona bene su cluster ma è ottimo per rappresentare piccoli record con relazioni complesse
    - Ogni nodo ha relazioni indipendenti con altri nodi. Le relazioni hanno nomi come PURCHASED, PAID_WITH o BELONGS_TO; questi nomi delle relazioni permettono di navigare nel grafo.
    - Supponiamo di voler trovare tutti i *Customers* che hanno acquistato un prodotto chiamato *Refactoring Database*. Tutto ciò che dobbiamo fare è eseguire una query per il nodo del prodotto *Refactoring Database* e cercare tutti i *Customers* con la relazione PURCHASED in ingresso
    - Graph DBMS usa il metodo CRUD (Create, Read, Update, Delete)
    - Labelled Property Graphs
        - Nodi contengono proprietà (coppie <chiave,valore> che descrivono caratteristiche aggiuntive)
        - I nodi possono avere un tag con una o più label
        - Le istanze hanno direzione, nome , nodo d'inizio e nodo di fine.

- Schemaless Databases
    - Key-value permette di archiviare qualsiasi dato sotto ad una chiave
    - Document DB non ha restrizioni sulla struttura del documento da archiviare (ma di solito sono simili)
    - Column-family DB permette di archiviare qualsiasi dato sotto qualsiasi colonna
    - Graph DB permette liberamente di aggiungere nuovi archi e proprietà ai nodi ed agli archi
    - Vantaggi:
        - Più libertà e flessibilità
        - Si può facilmente cambiare l'organizzazione dei dati
        - Si possono gestire dati non uniformi
    - Svantaggi
        - Presenza di schemi impliciti che spostano la responsabilità alle applicazioni
        - L'assenza di schema non assicura la consistenza (se ad esempio più applicazioni accedono allo stesso dato) e non permette di migliorare l'efficienza

- Materialized view
    - Una relational view è una tabella risultante dalla computazione di tabelle base (tipo join)
    - Una materialized view è una view precomputata e memorizzata su disco.
    - Efficace per dati che vengono letti frequentemente (evitando di ripetere calcoli complessi)
    - Costruzione di una materialized view con approccio eager:
        - La view materialized viene aggiornata allo stesso tempo della base di dati (se si aggiunge un dato viene ricomputata la view)
        - Utile se le letture sono molto maggiori delle scritture
    - Costruzione di una materialized view con approccio detached
        - Dei job batch aggiornano la view ad intervalli regolari
        - Buono quando non si vuole applicare overhead ad ogni aggiornamento di DB

- Distribuzione dei dati
    - Server singolo: Esegui il database su una macchina singola che gestisce tutte le letture e scritture sull'archivio dei dati (se si può la scelta migliore è nessuna distribuzione)
    - Sharding: distribuisce dati differenti su nodi differenti
        - Nel caso ideale, abbiamo utenti diversi che comunicano con nodi server differenti. Ogni utente deve interagire con un solo server, ottenendo risposte rapide da quel server (bisogna avvicinarsi il più possibile a questo caso).
        - Il punto principale degli aggregati è che li progettiamo per combinare dati che vengono comunemente accessi insieme (quindi gli aggregati diventano un'unità ovvia per la distribuzione).
        - Un altro fattore è cercare di mantenere il carico bilanciato.
        - **auto-sharding**: il database si occupa della responsabilità di allocare i dati sugli shard e garantire che l'accesso ai dati avvenga sullo shard corretto.
        - Lo shard da solo non migliora la resilienza.
    - Replication: copia stessi dati su più nodi.
    - Replicazione Master-Slave
        - Un nodo è designato come master, o primario. Questo master è la fonte autorevole per i dati ed è solitamente responsabile dell'elaborazione di eventuali aggiornamenti a tali dati. Gli altri nodi sono slave, o secondari. Un processo di replicazione sincronizza gli slave con il master

        - Utile da scalare per dataset a prevalenza di letture; è possibile gestire più richieste di lettura aggiungendo nodi slave e garantendo che tutte le richieste di lettura avvengano negli slave
        - Limitazione nell'aggiornare gli slave dopo scritture sul master (quindi da evitare per dataset con molte scritture)
        - Read resilience: se il master dovesse fallire, gli slave possono comunque gestire le richieste di lettura.
        - Utile avere slave come repliche del master così da sostituire quest'ultimo in caso di necessità
        - Problema di inconsistenza: C'è il rischio che i diversi client, leggendo da slave diversi, vedano valori differenti perché i cambiamenti non sono stati ancora propagati a tutti gli slave.
    - Replicazione Peer-to-Peer
        - Tutti i nodi hanno lo stesso peso, possono tutti accettare scritture e la perdita di uno di essi non impedisce l'accesso al data store
        - Con un cluster di replicazione peer-to-peer, puoi superare i guasti dei nodi senza perdere l'accesso ai dati. Inoltre, puoi aggiungere facilmente nodi per migliorare le prestazioni.
        - Problema di consistenza: scrivere su due luoghi diversi, corri il rischio che due persone tentino di aggiornare lo stesso record nello stesso momento (conflitto di scritture). Le scritture incoerenti sono permanenti (non come le letture).
    - Combinazione di sharding e replicazione.
        - La replicazione e lo sharding sono strategie che possono essere combinate. Se utilizziamo sia la replicazione master-slave che lo sharding, significa che abbiamo più master, ma ogni elemento di dati ha un solo master.
        - L'uso della replicazione peer-to-peer e dello sharding è una strategia comune per i database a colonne. In uno scenario come questo, potresti avere decine o centinaia di nodi in un cluster con i dati suddivisi su di essi.

- Persistenza: non c'è perdita di dati quando il sistema viene arrestato

- Disponibilità: i dati sono sempre disponibili quando necessari (ad esempio con un server backup), ma non è detto che contengano le scritture più recenti

- Tolleranza alla partizione: il sistema continua ad operare nonostante ci sia un problema di comunicazione che separa i cluster in partizioni che non sono in grado di comunicare tra di loro.

- Problemi di Consistenza
    - Read-Read conflict
        - Utenti diversi vedono dati diversi allo stesso tempo (mancato aggiornamento in scrittura da parte di un server)
        - Viola la replication consistency (stesso elemento ha lo stesso valore quando letto da repliche diverse)
    - Write-Write conflict
        - Due scritture sullo stesso elemento (risolvibile con i lock o via *version conflict*)
    - Read-Write conflict
        - Una lettura in mezzo a delle scritture correlate
        - Viola la logical consistency

        - Inconsistency window: La durata del periodo in cui l'incoerenza è presente

- BASE
    - Basycally Abailabe: fallimenti parziali sono possibili; il resto del sistema funziona
    - Soft state: i dati possono essere sovrascritti da dati più recenti
    - Eventual consistency: i dati possono essere temporaneamente inconsistenti

- Tipi di Eventual Consistency
    - Causal consistency:
        - il database riflette l'ordine in cui le operazioni sono aggiornate
        - ad esempio se aggiorno il conto a 10 e poi a 20 tutte le copie sono aggiornate a 10 prima di riaggiornarle a 20
    - Read-your-writes consistency
        - Quando un record è aggiornato, tutte le letture di tale record restituiscono il valore aggiornato
    - Session consistency
        - È una read-yout-write consistency durante una singola sessione
        - Se la sessione finisce e se ne inizia un'altra con lo stesso server non c'è garanzia che si ricordi delle scritture fatte nella sessione precedente
    - Monotonic read consistency
        - Se si esegue una query e si vede il risultato, non si vedrà mai una versione precedente
        - Alice aggiorna a 10 e Bob a 20, se Bob esegue la query vedrà 20 anche se i server non hanno ancora aggiornato i propri valori.
    - Monotonic write consistency
        - Gli aggiornamenti vengono eseguiti nell'ordine in cui sono stati emessi

- CAP Theorem (Consistency, Availability, Partition tolerance)
    - Dati i tre principi di Consistenza, Disponibilità e Tolleranza alle Partizioni, è possibile ottenere solo due di questi tre.
    - Un sistema a singolo server è l'esempio ovvio di un sistema CA: un sistema che ha Consistenza e Disponibilità, ma non Tolleranza alle partizioni. Un singolo server non può essere partizionato, quindi non deve preoccuparsi della tolleranza alle partizioni. C'è solo un nodo, quindi se è attivo, è disponibile. Essere attivi e mantenere la consistenza è ragionevole. Questo è il mondo in cui vivono la maggior parte dei sistemi di database relazionali.
    - In un cluster partizionato bisogna scegliere tra AP o CP (AC è meno responsivo)
    - Nel teorema CAP, la disponibilità viene definita come "ogni richiesta ricevuta da un nodo che non fallisce nel sistema deve produrre una risposta". Quindi, un nodo fallito e non reattivo non implica una mancanza di disponibilità secondo CAP
    - Se sacrifichiamo la tolleranza, significa che dovremo centralizzare tutto.
    - Se sacrifichiamo la disponibilità, i servizi rimangono non disponibili finché la consistenza non viene restaurata
    - Se sacrifichiamo la consistenza, bisogna accettare l'eventual consistency
    - La consistenza diventa la linearizzabilità. La disponibilità è "ogni richiesta ricevuta da un non-failing nodo deve ritornare una non-error risposta". La Partition Tolerance è "il sistema continua a funzionare anche se alcuni nodi non sono raggiungibili".
        - Linearizzabilità: significa che se un'operazione B è iniziata dopo l'operazione A completata correttamente, allora l'operazione B deve vedere il sistema nello

stesso stato lasciato dal completamento dell'operazione A. Per assicurare la linearizzabilità il sistema deve far apparire il sistema al cliente come una singola copia dei dati. La linearizzabilità è costosa.
    - CAP Availability: se avviene una partizione, ci sono due scenari possibili
        - Entrambi i data centers continuano ad operare, senza propagare i write (violando la linearizzabilità, ma mantenendo la CAP availability)
        - Le read e le write vanno al nodo leader, gli altri nodi smettono di funzionare (CAP-Availability non viene mantenuta, ma solo la Linearizzabilità).

- 2-Phase commit
    - I dati vengono salvati su diversi nodi e i dati vengono resi accessibili al cliente solo nel momento in cui la decisione di tutti i cluster è stata resa nota. Così ogni nodo sa se tutti hanno salvato il risultato o se hanno fallito.
        - STEP 1: ad ogni nodo viene chiesto di salvare i dati
        - STEP 2: l'update viene effettivamente portato avanti
    - Se un nodo non riesce a completare l'update, allora il coordinatore chiede a tutti i nodi di fare roll back e la transazione viene abortita.

- Quorum
    - Per mantenere una forte consistenza nella replica, non è necessario contattare tutti i nodi replicati, ma solo un quorum sufficientemente grande.
    - Se si verificano scritture in conflitto, solo una può ottenere la maggioranza. Questo concetto è noto come **write quorum** ed è espresso con la diseguaglianza $W > \frac{N}{2}$, ove il numero di nodi che partecipano alla scrittura (W) deve essere superiore alla metà del numero totale di nodi coinvolti nella replica (N).
    - **read quorum**: quanti nodi bisogna contattare per essere certi di ottenere l'ultima modifica aggiornata. Siano R la relazione tra il numero di nodi da contattare per una lettura, W il numero di nodi che confermano una scrittura, e N il fattore di replica si può ottenere una lettura fortemente consistente se $R + W > N$.
        - Consideriamo un fattore di replica pari a 3. Se tutte le scritture devono essere confermate da due nodi $(W = 2)$, allora dobbiamo contattare almeno due nodi per essere certi di ottenere i dati più recenti. Se, invece, le scritture vengono confermate solo da un nodo $(W = 1)$, dobbiamo contattare tutti e tre i nodi per assicurarci di avere gli ultimi aggiornamenti.

- Cypher (linguaggio query per Property Graph)
    - Cypher ha una sintassi intuitiva, basato sulla rappresentazione ASCII. Le query in Cypher non sono annidate ma **concatenate**, possono essere utilizzati diversi MATCH-WHERE-WITH, dove l'ordine delle clausole impatta sulla semantica delle query
        - Conta tutte le relazioni nel database:
          MATCH ()-[relationship]->()
          RETURN count(relationship) AS count
        - Trova tutte le coppie di fratelli
          MATH (parent)-[:HAS_CHILD]->(child1), (parent)-[:HAS_CHILD]->(child2)
          WHERE id(child1) <> id(child2)
          RETURN child1, child2

      Le query possono accedere sia ai labels (stringhe assegnate ai nodi) che alle properties (di nodi e relazioni)

        - Trova gli amici di tutte le persone con nome *Antonio*, e restituisci i loro nomi e l'inizio dell'amicizia (*Human* è una etichetta mentre *name* e *startDate* sono property keys)
          MATCH (:Human {name: `Antonio`})-[rel:HAS_FRIEND]->(friend:Human)
          RETURN friend.name, rel.startDate
- I Nodi sono indicati con le parentesi (), dove opzionalmente vengono inserite il nome della variabile tra backtick, una lista di labels (precedute da :) e un set di proprietà (una lista separate da virgole in {...})
- Sequenza di uno o più nodi, separati dalle seguenti espressioni:
    - -[...]-> (forward)
    - <-[...]- (backward)
    - -[...]- (bidirectional)

  Le frecce indicano quindi la direzione. L'espressione [...] indica una relazione, dove all'interno può essere inserito una variabile, un tipo di relazione, o un range ("*" indica che la relazione deve occorrere più volte)
    - Trova tutti i path che connettono due nodi con nomi Vincenzo e Gennaro:
      MATCH p=({name=`Vincenzo`}) –[*]->({name=`Gennaro`})
      RETURN relationships(p)
- In Cypher viene fatto il "Graph pattern matches", ossia il match di parte del pattern con parte del grafico. Ogni path pattern viene quindi mappato in un'alternanza di nodi e relazioni. Una query può utilizzare diverse clausole per permettere allo stesso edge di essere utilizzato più volte.
    - Tutte le persone che hanno una figlia con una relazione in informatica:
      MATCH (p:Person)-[:HAS_DAUGHTER]->()
      MATCH (p:Person)-[r]-({occupation:`computer scientist`})
      RETURN p.name, type(r)
- Con WHERE possiamo usare qualsiasi espressione, ad esempio operatori booleani (AND, OR, XOR and NOT), operatori matematici (+,*, sin, ceil) e funzioni su stringhe (substring e toLower). È possibile utilizzare funzioni di aggregazione come MIN, AVG, COUNT e anche quelle più complesse come stDev.
- A differenza di SPARQL e SQL, il raggruppamento è implicito: se vengono definiti risultati aggregati e non aggregati, quelli non aggregati saranno le chiavi in base alle quali effettuare il raggruppamento.
- DISTINCT viene utilizzato per ridurre i risultati così da non contenere duplicati prima che vengano utilizzate funzioni di aggregazione.
- Con il WITH siamo in grado di concatenare una subquery (invece di annidarla). Può essere seguita da ORDER BY (ordina i risultati, può essere utilizzato DESC o ASC), LIMIT (prende i primi n risultati) e SKIP (salta i primi n risultati).
- UNION viene utilizzato per combinare i risultati di due query, rimuove automaticamente i duplicati.
- OPTIONAL MATCH è usato per aggiungere informazioni se disponibili. Se il match non viene trovato, la variabile viene mappata come null.
    - L'uso di null in Cypher è ambiguo (ordinamento e comparazione di valori nulli non sono ancora stati definiti in maniera consistente.
- Funzioni
    - all(*variable* IN *list* WHERE *predicate*) restituisce true se il predicato risulta vero per tutti gli elementi della lista.
    - any(*variable* IN *list* WHERE *predicate*) restituisce true se almeno per un elemento della lista è vero il predicato.

        - exists(*pattern*) restituisce true se il match per un determinato path nel grafo esiste.
        - isEmpty(*list*) restituisce true se la lista non contiene elementi o la STRING data non contiene caratteri
        - none(*variable* IN *list* WHERE *predicate*) restituisce true se il predicato non è vero per nessun elemento della lista.
        - single(*variable* IN *list* WHERE *predicate*) restituisce tue se il predicato è soddisfatto esattamente da un solo elemento nella lista
    - Quantificatori

Variant	Canonical form	Desc.
{m,n}	{m,n}	*m* to *n*
+	{1,}	≥ 1
*	{0,}	≥ 0
{n}	{n,n}	*m* to *n*
{m,}	{m,}	$\geq m$
{,n}	{0,n}	≥ 0 and $\leq n$
{,}	{0,}	≥ 0

        - Vengono utilizzati nel seguente modo: ((Nodo) -[...]- (nodo)) {quantificatore}
    - UNWIND è una clausola utilizzata per trasformare una lista in righe individuali, nel seguente modo UNWIND <*list_expression*> AS <*variable*> .
        - La seguente query produce una riga per ogni elemento della lista, null incluso.
          UNWIND [1, 2, 3, null] AS x
          RETURN x, `val` AS y
    - Collect() è una funzione di aggregazione che raggruppa i valori in una lista, in base ad una chiave (ad esempio i film per ogni attore)

- RDF (Resource Description Framework)
    - Gli RDF sono un modello di rappresentazione dati basato su triple:
        - **Soggetto**: l'entità di cui si parla
        - **Predicato**: la proprietà o la relazione
        - **Oggetto**: il valore o un'altra entità collegata

      [Alice] -- conosce --> [Bob]

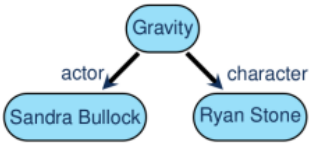
  Esistono vari formati sintattici:
    - **N-Triples**
        - È il più semplice formato concepibile: ogni linea codifica una tripla e termina con un punto "."
        - Gli IRI (Internationalized Resource Identifier) si scrivono tra parentesi angolai "< >"
        - Le stringhe si scrivono tra virgolette "" ed i commenti iniziano con #
        - I bnode (blank node) sono definiti con *_:stringId* (esempio "_:3"

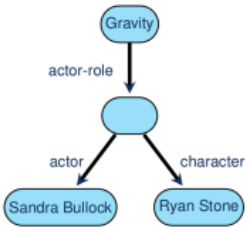
      <http://esempio.org/Alice> <http://xmlns.com/foaf/0.1/knows> <http://esempio.org/Bob> .
      <http://esempio.org/Alice> <http://xmlns.com/foaf/0.1/name> "Alice" .
      <http://esempio.org/Bob> <http://xmlns.com/foaf/0.1/name> "Bob" .
        - Veloce e facile da analizzare ed è processabile anche da tools base come grep
        - Molto pesante a livello di spazio e non particolarmente human-friendly
    - **Turtle**
        - Estende N-Triples con delle abbreviazioni
        - Dichiarazione di prefissi e namespace permettono di ridurre gli IRI
        - Se le triple terminano con ";" allora si assume che la prossima tripla inizia con lo stesso soggetto
        - Bnode sono codificate con le parentesi quadre "[]"

        - BASE è usato per dichiarare un IRI base:
          BASE https://example.org/
        - PREFIX è usato per dichiarare abbreviazione per prefissi IRI
          PREFIX xsd: <https://www.w3.org/2001/XMLSchema#>
        - Molto semplice e facile da processare e se formattato con attenzione anche Human-readable
        - Non è processabile in modo sicuro da grep o tools simili
    - **RDF/XML** utilizza un encoding basato su XML, diventa meno leggibile e non è in grado di fare l'encoding di tutti i grafi RDF
    - **JSON-LD** encoding basato su file JSON e specifica come il JSON si mappa su RDF.
    - **RDFa** utilizza un embedding HTML delle triple, principalmente utilizzato per annotazioni di documenti.

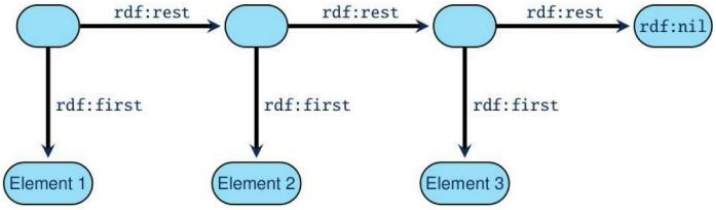
Il goal principale degli RDF è di permettere lo scambio di dati tra applicazioni, attraverso gli IRI. Non tutti i dati possono essere in RDF (file media, eseguibili, ecc.) ma molti si (come dati relazionali, documenti XML, grafi, etc...)

- Reification
    - Viene introdotto un nodo ausiliare per rappresentare la relazione:

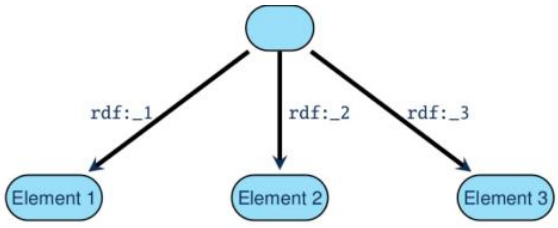


diventa


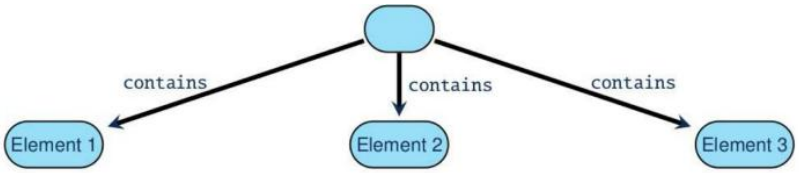
    - La reification sulle triple è il metodo principale per passare da un DB relazionale ad uno RDF
- Essendo gli RDF triple non ordinate per rappresentare e liste ordinate ci sono diversi metodi:
    - Utilizzare le linked list che sono supportate da RDF collections:



      Svantaggio: richiede troppi step per gli accessi
    - Rappresentare l'ordine attraverso le proprietà:



      Accesso più diretto ma risulta difficile gestire insert e delete
    - Ignorare l'ordine e rappresentare la lista come un insieme:



- SPARQL (pronuncia: “sparkle”)
  - Linguaggio standard per manipolare grafi RDF
  - Esempio di query che fa la lista di tutte le risorse IRI con labels
    PREFIX rdfs: <http:/www.w3.org/2000/01/rdf-schema#>
    SELECT ?resource ?label
    WHERE {
    ?resource rdfs:label ?label .
    }
  - Le query consistono nei seguenti blocchi:
    - Prologo: per dichiarazioni PREFIX e BASE (funzionano come in Turtle)
    - Clausola Select: SELECT seguita da una lista di variabili (es. ?person) e assegnamenti (es. (COUNT (*) AS ?count))
    - Clausola Where: WHERE seguito da un pattern
    - Modificatori di insieme: come LIMIT e ORDER BY
  - SPARQL supporta altre forme di query
    - ASK: per vedere se ci sono dei risultati (ma non li restituiscono)
    - CONSTRUCT: per fare un grafo RDF dai risultati della query
    - DESCRIBE: per avere un un grafo RDF con info aggiuntive.
  - Gli IRIs sono abbreviati usando i nomi dichiarati in PREFIX o dichiarati in BASE.
  - Le variabili sono stringhe che iniziano con ? (si può usare anche $, ma è meno usato).
  - Pattern in SPARQL
    - Un triple pattern è una tripla <*soggetto, predicato, oggetto*>
    - Un BGP (basic graph pattern) è un insieme di triple pattern
    - I BGP vengono interpretati congiuntamente, cioè si cerca una corrispondenza che soddisfi tutte le triple contemporaneamente.
  - Significato dei blank node nei query pattern
    - Denotano una risorsa non specificata
    - Sono simili alle variabili ma non possono essere usati in SELECT
    - Per i bnode si usa la sintassi Turtle (*[]* oppure *:nodeId*)
  - Significato dei blank node nei risultati delle query
    - Indicano che una variabile ha trovato corrispondenza con un bnode nei dati
    - Gli ID nei risultati sono **ausiliari** e possono differire dagli ID dei dati
  - Risultati delle query SPARQL
    - Una solution mapping (μ) è una funzione parziale dai nomi di variabili a termini RDF. Una solution sequence è una lista di solution mappings
    - Senza ordinamento, le soluzioni formano un multiset (insieme con elementi ripetuti)
    - Dato un grafo RDF G e una BGP P, μ è la soluzione se è definita esattamente sulle variabili presenti in P e esiste una mappatura σ formata di blank nodes ai termini RDF tali che $\mu\sigma(P) \subseteq G$
    - la cardinalità di μ nel multiset è il numero di σ distinti che soddisfano la condizione
  - Paths in SPARQL
    - I Knowledge graphs riguardano le connessione indirette e i Property paths vengono utilizzate per specificare le condizione su di esse
    - Il prefisso ^ viene utilizzato per cambiare la direzione della relazione, mentre | viene usato per indicare l’alternativa.
      WHERE {
      eg:JSBach (^eg:hasFather | ^eg:hasMother)+ ?descendant.
      }

- Clausole
  - SELECT specifica quali variabili ritornare, potrebbe calcolare delle aggregazione direttamente nel select (ad esempio il count). Può essere utilizzato SELECT DISTINCT per porre la molteplicità ad 1.
  - Si possono usare ORDER BY, LIMIT e OFFSET (stesso di skip di Cypher). Vengono posti sempre dopo il WHERE.
- Funzioni di aggregazione (tutte accettano DISTINCT per rimuovere i duplicati)
  - GROUP BY separa le soluzioni in gruppi in base alla Key.
  - COUNT, SUM, AVG, MIN, MAX sono tutte disponibili.
  - SAMPLE prende un valore random
  - GROUP_CONCAT concatena le stringhe in ordine.
  - HAVING filtra le risposte prodotte dall'aggregazione
  - FILTER viene utilizzato per tutte le espressioni SPARQL che non sono basate sul grafico RDF (ad esempio confronti numerici, string matching). Non producono nuove risposte, ma filtra le risposte eliminando quelle che non soddisfano la condizione. Può essere usato in combinazione con NOT EXIST per testare l'assenza del pattern nel grafo.
- Gruppi
  - Una parte di query tra parentesi graffe {} è chiamata gruppo in SPARQL
  - Semanticamente, i risultati in un gruppo sono combinati con una Join
- Altri operatori
  - UNION è utilizzato per ottenere l'unione di due risultati (potrebbe essere necessario usare DISTINCT).
  - MINUS rimuove i risultati di un gruppo da quello di un altro.
  - OPTIONAL è utilizzato per estende i risultati del mapping ed è usato all'interno di un gruppo insieme al triple pattern.
  - VALUES è utilizzato per assegnare un valore costante ad una variabile. Il risultato è come un insieme di subqueries o di unione di query annidate, ma è molto più efficiente.
  - BIND è per assegnare un valore computato ad una variabile.

- Wikidata
  - Wikidata è un knowledge graph RDF con un DB immenso (più di 100M entità) costruito dalla community e restituisce un mapping RDF
  - Contenuto dei documenti delle entità
    - **Entity ID**: identificatori indipendenti dalla lingua.
    - **Intestazione dei termini**: e pagine dei documenti iniziano con un'etichetta, una breve descrizione e un elenco di alias nella lingua dell'utente (o nella lingua migliore disponibile); i termini possono essere inseriti in diverse centinaia di lingue e sistemi di scrittura.
    - **Statements**: la parte principale della pagina consiste in affermazioni basate sulla fonte, relative a varie proprietà che un'entità può avere; le dichiarazioni possono avere un *rank* (normale, preferito, deprecato) per indicare la loro rilevanza attuale.
    - **Site links**: collegamenti a pagine su altri progetti Wikimedia che realizzano l'integrazione delle informazioni a livello di entità.
  - Come RDF, wikidata è basato su multi-grafi orientati ed etichettati, e le proprietà hanno le loro identità.

- A differenza di RDF, gli statemens in wikidata possono avere annotazioni e reference e wikidata ID non sono immediatamente IRI (sono composti dall'entity ID e dal page ID)

- Datalog
  - Datalog è un linguaggio di interrogazione per basi di dati che si presenta come un linguaggio di programmazione logica derivante da Prolog relativo ai database relazionali. Come in logica si basa su regole di deduzione, ma non permette l'utilizzo di simboli di funzioni né un modello di valutazione non procedurale.
  - **Complessità**: Dati un Programma P e un database D,
    - la Data Complexity (dove P è fisso e D è parte dell'input) è espresso come PTIME-completo.
    - Combined Complexity (sia P che D sono parti dell'input) è EXPTIME-completo.
  - **Sintassi**: Una regola datalog è una espressione della forma: $r_0(X_0)$:-$r_1(X_1), \dots, r_n(X_n)$
    - Ogni regola è composta da una testa (conseguente) e da un corpo(antecedente), a loro volta formati da uno o più predicati atomici. Se tutti gli atomi del corpo sono verificati, ne consegue che anche il predicato atomico della testa lo sia.
    - $n \geq 0$ (il corpo può essere vuoto)
    - $r_0, \dots, r_n$ sono nomi di relazioni
    - $X_0, \dots, X_n$ sono tuple di variabili (ad es. $X_0 = A, B, C$)
    - Tutte le variabili in testa (X e Y) devono apparire nel corpo
    - Gli argomenti dei predicati sono i termini (variabili o costanti)
  - Sia P un programma datalog (insieme finito di regole) è possibile descrivere un dominio attraverso predicati estensionali (quelli che non occorrono nella testa di una regola di P), che corrispondono alle relazioni, e predicati intensionali (occorrono in testa di alcune regole di P) che vengono specificati attraverso regole logiche e che ne arricchiscono il modello. $EDB(P)$ è l'insieme delle relazioni estensionali di P, mentre $IDB(P)$ l'insieme delle relazioni intensionali di P. Una datalog query Q è una coppia $P, answer$ dove P è il programma e $answer$ una relazione intensionale chiamata relazione di output.
    - Esempio: Tutte le città sono raggiungibili da Vienna
    - Programma P:

      Reachable(x,y):-Flight(x,y,z)
      Reachable(x,q):-Flight(x,y,z),Reachable(y,w)
      Answer(z) :- Airport(x,Vienna),Airport(y,z),Reachable(x,y)
    - $EDB(P) = \{\text{Flight,Airport}\}$
    - $IDB(P) = \{\text{Reachable,Answer}\}$
    - $Q = (P, \text{Answer})$
  - Datalog si è dimostrato essere più espressivo dell'algebra relazionale, data la possibilità di scrivere regole ricorsive, in grado di richiamare il medesimo atomo della testa anche nel corpo della regola. Questo porta però alla possibilità di scrivere regole indecidibili, infatti per preservare l'integrità della base di dati vengono adottate alcune regole chiamate "**safety conditions**":
    - I predicati estensionali possono comparire solo nel corpo delle regole, così da non poter ridefinire le relazioni memorizzate nella base di dati.
    - Tutte le variabili che appaiono nella testa devono apparire anche nel corpo della regola (questo garantisce che il processo di deduzione giunga al termine)
  - **Semantica model-theoretic**
    - Il model-theoretic cerca di definire il significato di un programma trovando i modelli logici che soddisfano tutte le sue regole. Il modello minimo sarà il sottoinsieme più piccolo che soddisfa tutte le regole

- Il problema è che non ci dice come calcolare il significato, sappiamo solo che può essere ottenuto come intersezione di tutti i modelli.
- n generale si segue la seguente regola per costruire il modello:
  Un'interpretazione I è un modello del programma P se, per ogni regola, vale:
  Se il corpo della regola è vero in I, allora anche la testa è vera in I.

- **Semantica fixed-point**
  - Dati un programma P e un database D, l'operatore di conseguenza immediata è una mappatura T da D a D, che aggiunge tutti i nuovi atomi ground che vengono derivati dalle regole del programma in un singolo step.
  - Sia ad esempio $p(X,Y) \leftarrow r(X,Z), s(Y,a,X)$ dove $r(b,d)$ e $s(c,a,b)$ sono atomi in D. Allora la conseguenza immediata di $D\ wrt\ P$ è $p(b,c)$, ossia quello che ne deriviamo dalla regola.
  - Si indica con $T_P(D)$ l'operatore di conseguenza immediata. T_P ha anche un minimo punto fisso (o least fixpoint) che contiene D e rappresenta il significato del programma (la semantica), indicato come $P(D)$
  - $T_{P,0}(D) = D; \quad T_{P,i+1}(D) = T_P\left(T_{P,i}(D)\right); \quad T_{P,\infty} = \bigcup_{i=0}^{\infty} T_{p,i}(D)$
  - $T_{P,\infty}$ rappresenta il minimo punto fisso di T_P contenente D
  - Coincide con il modello minimo della semantica model-theoretic ma a differenza di quest'ultimo descrive un algoritmo per ottenere la semantica (può essere inefficiente se non ottimizzato)
- **Semantica proof-theoretic**
  - Descrive la semantica in termini di dimostrazioni, cioè un fatto sussiste se esiste una derivazione finita che lo prova a partire dai fatti iniziali. In questo caso si usano regole di inferenza e un fatto viene visto come vero se esiste una prova finita a partire dai fatti noti.
  - Utile per spiegare il perché un fatto è vero
  - Equivalente agli altri due approcci (per Datalog positivo, senza negazione).
  - Contro: Meno adatto per l'implementazione diretta (più teorico).
- Le regole sono frasi logiche che indicano una proprietà dell'output. Convertiamo il programma P in un set di frasi di Logica del primo ordine universali che chiamiamo $\Sigma[P]$, che descrive l'output di P su D. Possiamo avere infiniti modelli, quindi bisogna specificare quale sarà l'output.
- Un database D è un modello della frase $\phi[\rho]$, scritta come $D \vDash \phi[\rho]$ se in qualsiasi momento esista l'omomorfismo h su $\phi[\rho]$ a D, allora h sarà omomorfismo anche nella conclusione
  - $\phi[\rho]$ è una frase del tipo $\forall V_1, \dots, \forall V_n\left(r_1(X_1) \wedge \dots \wedge r_n(X_n) \rightarrow r_0(X_0)\right)$ con V_i variabili che occorrono nella regola ρ
  - L'omomorfismo h: $terms(A) \rightarrow terms(B)$ è una sostituzione che assegna valori costanti alle variabili della regola in modo che tutti gli atomi del corpo della regola, quando applicata h, sono fatti contenuti nel database D.
- Il modello intenzionale.
  - Il modello intenzionale non deve contenere più atomi di quelli necessari per soddisfare $\Sigma[P]$. Scegliamo infatti il modello minimale che contiene il seguente database: $MM(P,D) = \{D' : D \subseteq D', D' \vDash \Sigma[P]\}$
  - $MM(P,D)$ contiene esattamente un database. Infatti, se uniamo D ad una nuova relazione con termini presenti all'interno del database, allora sarà un modello di $\Sigma[P]$ (questo significa che $|MM(P,D)| \geq 1$); mentre, dati n Database che

appartengono al modello minimale, allora l'intersezione dei database apparterrà al modello minimale $MM(P, D)$ (ovvero $|MM(P, D)| \leq 1$)

- Come calcolare l'output
  - Per la **Semantica Model-theoretic**: $M(P, D) = D' : D \subseteq D', D' \vDash \Sigma[P]$. Quindi assumendo di avere $M(P, D) = D_1, \dots, D_n$ abbiamo che il modello minimale verrà calcolato come segue: $MM(P, D) = D_1 \cap \dots \cap D_n$
  - Per la **Semantica fixed-point**: dato un database D e una query $Q = (P, \text{Answer})$, definiamo l'output di P su D, attraverso un operatore che calcola gli atomi che possono essere derivati da D e P e il cui output è il minimo punto fisso dell'operatore. Questo operatore è l'operatore di conseguenza immediata T_p. L'output di P su D, indicato come $P(D)$, è il minimo punto fisso di T_P contenente D. Questo verrà quindi calcolato facendo l'unione dei singoli $T_{P,i+1}$
- Datalog (standard, senza estensioni) non può esprimere query che richiedono:
  - Negazione non stratificata (o con cicli negativi)
  - Aggregazioni (COUNT, SUM, MAX…)
  - Ordine (TOP-K, MIN, MAX con ordinamento)
  - Query non monotone (che richiedono ricordare fatti)
  - Contare o confrontare cardinalità (es: "più di 3 amici")