



UNIVERSITÀ DEL SALENTO
DEPARTMENT OF INNOVATION ENGINEERING

MASTER'S DEGREE IN COMPUTER ENGINEERING

DATABASE

Database

Dott. Marco Chiarelli

Academic Year 2016/2017

Quest'opera è stata rilasciata con licenza Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 3.0 Unported. Per leggere una copia della licenza visita il sito web <http://creativecommons.org/licenses/by-nc-sa/3.0/> o spedisci una lettera a Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.



Questi appunti sono stati scritti utilizzando L^AT_EX tramite la distribuzione MiK_TE_X <http://miktex.org/>

Come editor è stato usato TeXMaker 4.5 <http://www.xm1math.net/texmaker/>

Dei contenuti rielaborati in questa opera, salvo esplicitamente scritto il contrario, il prof. Mario Alessandro Bochicchio non se ne assume alcuna responsabilità.

Tali contenuti sono stati scritti INTEGRALMENTE dagli studenti di Ingegneria Informatica at UNISALENTO, II anno 2016/2017. Marco Chiarelli, seppure l'unico redattore di tale manodopera, NON ne è in alcun modo il suo autore principale.

Indice

Introduzione	ii
1 Modeling	1
1.1 Progettazione	1
1.1.1 Caso di Studio - Google	1
1.1.2 Organizzazioni	3
1.1.3 Lo scenario della progettazione - Progetto e realizzazione	5
1.1.4 Scenario Ingegneristico della Progettazione	6
1.2 Architettura di un Progetto Software	7
1.2.1 L'hardware	7
1.2.2 Il software - Architettura a tre livelli	9
1.3 Modellazione dei dati	10
1.3.1 ESEMPIO – APPLICAZIONE CLIENT-SERVER	11
1.3.2 ESEMPIO	13
1.3.3 ESEMPIO - THE COMPANY DATABASE	14
1.4 Conceptual Model	17
1.4.1 Il valore NULL	19
1.4.2 ANALIST	19
1.4.3 SCHEMA	20
1.4.4 Come iniziare la progettazione di un DB	22
1.4.5 Ternary relation type	22
1.5 APPENDICE	24
1.5.1 DESIGN PATTERN	24
1.6 Esercizio - Fermate Autobus	25
1.6.1 Design Pattern: Preventivo-Consuntivo	32
1.6.2 Conclusione	33
1.7 DATA CENTRIC APPLICATION	34
1.7.1 USER TYPES	34
1.7.2 ESEMPIO	35
1.7.3 SIMPLIFIED MAPPING ALGORITHM	35
1.7.4 PRESENTATION LAYER FOR CUSTOMER OR DIRECTOR	36
1.7.5 BUSINESS RULE LAYER	38
1.8 LIST VIEW and DETAIL VIEW	39
1.8.1 DATA PATTERN	40
1.9 DBMS	54
1.9.1 Descrizione dei principali DBMS	54
1.9.2 Installazione e configurazione di MySQL	55
1.9.3 PRIMO CONTATTO CON MYSQL WORKBENCH	63
1.10 IMPORTANZA MODELLI	74

1.10.1	IMPORTANZA MODELLO DATI	74
1.10.2	IMPORTANZA INFORMAZIONE	75
1.11	DIAGRAMMI EER	76
1.11.1	SCHEMA EER CAPITOLO 4	79
1.11.2	ESERCIZIO SVOLTO IN CLASSE	79
2	MODELLO LOGICO	81
2.1	SQL QUERY: Le interrogazioni del database	81
2.1.1	ALGEBRA RELAZIONALE	82
2.1.2	Operatore Join	84
2.2	Sviluppo di una Web Application	86
2.2.1	XAMPP	86
2.2.2	MySQL Workbench	92
2.2.3	DataGrip	95
2.3	Algebra Relazionale	96
2.3.1	INTRODUZIONE	96
2.3.2	ALTRI OPERATORI	97
2.3.3	ESEMPI DI QUERY	100
2.4	Queries	101
2.5	Popolazione casuale DB	106
2.5.1	Tipi di dato in MySQL	106
2.5.2	Attributi	107
2.5.3	Funzioni di Aggregazione	107
2.5.4	Popolazione del DataBase	108
2.6	MySQL	113
2.6.1	Tipi di tabelle	113
2.6.2	AUTO-INCREMENT	117
2.6.3	Tipi di dato in SQL	118
2.6.4	Funzioni di aggregazione	123
2.6.5	phpMyAdmin – indici	127
2.6.6	Popolare database	128
2.7	SQL Recap	129
2.7.1	VINCOLI	131
2.7.2	Meccanismo base Query	131
2.7.3	ESERCIZIO	132
2.8	SQL per manipolare i dati	135
2.8.1	Transazioni e concorrenza (accenno)	137
2.8.2	Algebra a tre livelli	138
2.8.3	Query più complesse	139
2.8.4	Vincoli in SQL	144
2.8.5	Asserzioni	144
2.8.6	Trigger	145
2.8.7	Viste (Views)	145
2.8.8	Esercizio	146
2.9	Continuazione Popolamento DB	147
2.9.1	CRUD (Create, Read, Update, Delete)	150
2.10	CRUD cycle	165
2.10.1	PDO (PhpData Objects)	166
2.10.2	Grid per le operazioni CRUD	167

2.11 Enhanced Mapping	178
2.11.1 ALGORITMO DI MAPPING	179
2.12 Overview	181
2.12.1 Modellazione di dati	183
2.13 Mapping RECAP	185
2.13.1 Svolgimento traccia d'esame	188
2.14 Qualità di un DB	196
2.14.1 DIPENDENZE FUNZIONALI	199
2.15 Transazioni e Concorrenza	202
2.15.1 Transazioni	202
2.15.2 Concurrency Control Techniques	207
2.16 Indicizzazione e Indici	208
3 DATA WAREHOUSE	209
3.1 Introduzione	209
3.2 Analisi multidimensionale	210
3.2.1 Operazioni concettuali	213
3.3 Analisi Multidimensionale e Data Warehousing	214
3.4 COSTRUTTI AVANZATI DEL DFM	221
3.4.1 ESEMPIO DELLE VENDITE (DA E/R)	224
3.4.2 Passaggio dall'ER riconciliato al DFM: l'ALBERO DEGLI ATTRIBUTI	224
3.4.3 CARICO DI LAVORO	225
3.4.4 PROGETTAZIONE LOGICA	226
3.5 LE TABELLE PIVOT DI EXCEL PER IL DATA WAREHOUSE	227
3.5.1 QUERY	230
3.6 PROGETTAZIONE LOGICA	235
3.6.1 Indicatori	235
3.6.2 Motore Data Warehouse	237
3.6.3 SnowFlake Schema	237
Appendici	238
3.7 Esercizio (Core di Facebook)	238
3.7.1 Esempi di interrogazioni	242
3.7.2 Meta-Database e Meta-Dati	243
3.8 Modellazione	243
3.8.1 Stima della dimensione del database	244
3.8.2 MODELLO VISIBILITY	245
3.9 Esercizio (Compagnia Aerea)	246
3.9.1 Continuazione ESERCITAZIONE	250
3.9.2 DFM	253
3.10 DATABASE TECHNOLOGIES	255
3.10.1 CLOUD COMPUTING	255
3.10.2 NIST REFERENCE ARCHITECTURE	256
3.10.3 BIG DATA	257
3.10.4 CAP'S THEOREM	257
3.10.5 DATABASE NoSQL	258
3.10.6 MAP REDUCE	258

Ringraziamenti

Un grazie particolare va ai miei compagni d'università, Dino Sbarro, Gabriele Accarino, Giampiero D'Autilia, Matteo Settembrini, Paolo Panarese ed Emanuele Costa Cesari.

Introduzione

- **Prerequisiti**

Buona conoscenza di linguaggi *Object-Oriented* (almeno uno), tecniche e strumenti. Elementi di computer networks e tecnologie di Rete, Web;

- **Abilità acquisite**

Lo studente sarà in grado di progettare e capire i modelli dei dati, creare e gestire database e progettare ed implementare applicazioni data-centric.

Lo scopo è fornire le basi circa le principali teorie sui database, tecniche e strumenti per usare i database e **progettare/implementare** database **applications**.

Argomenti:

- Database, database relazionali, NoSQL e NewSQL;
- Sistemi di gestione dei database (DBMS);
- Modello Relazionale ed Algebra Relazionale;
- SQL: definizioni dei dati e loro manipolazioni;
- Basi della Computer-Human Interaction e progettazione delle interfacce;
- Aspetti architetturali: Clients, Servers, Peers, Dispositivi, IoT, ...
- Principi di Data-Analytics;
- Analisi multidimensionale e data-warehouse;

Capitolo 1

Modeling

1.1 Progettazione

Dal punto di vista ingegneristico i costi di progettazione di una Ferrari e di una Fiat non sono poi così differenti. La reale differenza risiede nel modo in cui i due differenti progetti sono ottimizzati per lo specifico target di clientela e per le caratteristiche ed aspettative, evidentemente differenti, delle due automobili. Un ingegnere progetta qualcosa di *customizzato*, cioè creato su misura per il committente, di fatto proiettando un'idea che risiede nella sua mente. A differenza di un progettista di automobili o di edifici, un progettista del software è in grado di interagire con la mente delle persone e di cambiarne le idee.

1.1.1 Caso di Studio - Google

Oggigiorno i Database sono ovunque. Google è uno dei DB più diffusi. Quando viene effettuata una ricerca in realtà viene eseguita una query per comunicare con il SW di Google. La successiva figura illustra un sistema ***Client/Server*** per interrogare un Database. Un **DBMS** (*Database Management System*) è un sistema che consente di gestire uno o più DB.

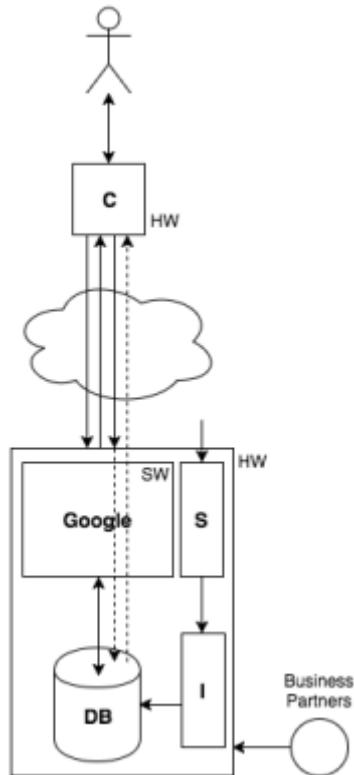


Figura 1.1: Architettura Client-Server

La nuvola rappresenta Internet, un insieme dinamico di connessioni, mentre il Client è una macchina (HW) al cui interno risiede il SW. Tramite l'inserimento dell'url nel browser viene generato un round-trip. Una volta stabilita la connessione con il DBMS sarà possibile accedere alle informazioni che risiedono nel DB. Ad esempio la ricerca delle parole: "Cinema Lecce" restituirà tutte le occorrenze dell'intersezione degli insiemi costituiti dai risultati di ricerca delle due singole parole. Sui computer di proprietà di Google oltre al DBMS esistono altri software: **Spider** o **Crawler** e **Indexer**.

Supponiamo che il web sia un grafo connesso, ovvero che ogni pagina web che risponde ad un url contenga i riferimenti (url) ad altre pagine web e così via, fornendo un url allo Spider esso ricorsivamente percorre tutto il grafo passando ciascuna pagina visitata all'Indexer. L'Indexer ha il compito di effettuare lo stamming delle parole ovvero la loro estrazione ed il mantenimento dei riferimenti alle pagine cui sono state trovate. L'idea pertanto consiste nel mantenere un indice analitico delle parole e i riferimenti a tutte le pagine web che le contengono. Questo consente di riuscire ad immagazzinare tantissimi riferimenti utilizzando pochi gigabytes. Quindi quando vogliamo effettuare una ricerca nel web ricorrendo ad un motore di ricerca le ricerche in realtà sono state già effettuate e l'unica cosa che fa il software è incrociare le informazioni. Google viene pagata da altre grosse società per far salire in cima i risultati delle ricerche per i contenuti che le riguardano. Osserviamo che l'ipotesi di rappresentazione del web attraverso un grafo connesso non è rappresentativa di alcune realtà come quella del deep web. Il deep web è costituito da isole cioè porzioni di grafo non connesse a quelle del web pertanto non raggiungibili sebbene nelle isole vengano utilizzati gli stessi protocolli del web (HTTP, ecc.).

1.1.2 Organizzazioni

Per quanto concerne le persone che hanno a che fare con un sistema SW, possiamo distinguere tre tipologie principali: {**User**, **Consumer**, **Customer**}. Per quanto riguarda le modalità di interazione e comunicazione con un sistema SW, troviamo la tipica architettura **Client-Server (C/S)** e la **Peer-to-peer (P2P)**. Nel P2P le entità in gioco sono tutte alla pari, nel senso che avvengono degli scambi alla pari di informazioni. Internet è oggi dominata da questi due meccanismi. Abbiamo cominciato a vedere una prima tipologia, gli user, che sono però una parte di tutto il resto: gli **stakeholders** che letteralmente sono dei portatori di interesse, qualcuno interessato ad un sistema. Gli **UT (User Types)** sono un sottoinsieme degli Stakeholders. Ad esempio, il rettore è uno stakeholder, ma non un utente. A volte sono proprio loro che comandano il tutto! Egli potrebbe pure non interagire con il SW, decide però le regole in gioco. Per parlare di uno stakeholder bisogna necessariamente precisare il sistema: per Google ad esempio, Ferrari è uno stakeholder particolare, denominato **business partner**. Parliamo ora di **Sistema** ed **Organizzazione**. Cos'è un'organizzazione? Esistono le persone e poi le organizzazioni. L'Università è un tipico esempio di organizzazione: struttura di persone che svolgono dei compiti ed hanno delle responsabilità, esse si occupano tipicamente di prodotti e di servizi. All'interno di un'azienda non vi può essere soltanto il singolo venditore, tipicamente vi è anche una fitta rete di assistenza. Noi abbiamo sempre a che fare con bundle di prodotti e servizi, ad esempio il comune telefono è un prodotto ed un servizio allo stesso tempo pertanto oggi quasi tutto è quindi un bundle di prodotti e servizi. Per schematizzare una tipica organizzazione possiamo avvalerci del modello di Anthony, che prevede una suddivisione piramidale dell'organizzazione in tre parti o livelli: Direzione, Management ed Operativa. Nell'Università ad esempio abbiamo il rettore, i presidi ed i professori e tecnici, stesso discorso per banche, ospedali, etc. La direzione è costituita da uno o pochi individui o da un consiglio di amministrazione, poi c'è il manager, il quale non è il capo dell'azienda: è il vice-capo ma risponde al direttore. I manager sono, nell'esempio dell'università, i direttori di dipartimento. Il personale operativo è chiamato anche **Front Office**. Svolgere i compiti operativi all'interno dell'università. Non bisogna però confondere Ruolo e Persona. L'organizzazione può essere rappresentata da un **organigramma**, la carta che rappresenta gli organi che compongono l'azienda e che rappresentano, tipicamente, una struttura gerarchica. Le organizzazioni, per essere ben guidate, a meno che non presentino strutture di tipo rete, devono essere strutturate in modo gerarchico, in alcuni casi vi potrebbe addirittura essere una replicazione del modello gerarchico. Nel caso delle holding ad esempio la struttura è di tipo gerarchico ed ogni organizzazione costituente è a sua volta una struttura gerarchica. Il sistema informativo esiste da sempre e non è costituito da bit! Esso è costituito da informazioni. Il sistema informatico è rappresentato da computer e dalle reti.

Sebbene le informazioni esistano da sempre il sistema informatico è piuttosto recente, infatti nel tempo può cambiare il processo di elaborazione delle informazioni, il quale insieme alle informazioni stesse costituisce il sistema informativo. Attualmente il sistema informativo esiste su sistemi informatici ma prima dell'avvento dei computer "viveva" sulla carta. Come si crea un sistema informativo per un particolare ruolo? Ad esempio, per un direttore, preside o professore. Il ruolo è svincolato dal soggetto fisico! Lo stesso Anthony, con la sua piramide, ci fornisce un grande supporto per l'attività di Analisi dei Requisiti. Sempre riferendoci alla struttura piramidale, possiamo individuare altri tre livelli che meglio identificano e raggruppano i differenti tipi di dati che vengono trattati dai ruoli che appartengono ai tre differenti livelli gerarchici: La **BI (Business Intelligence)**, l'**ERP (Enterprise Resource Planning)**, ed il **CRM (Customer Relationship Management)**. Nel primo livello troviamo tipicamente sistemi DSS (*Decision Support Systems*). Invece nello strato intermedio troviamo sistemi per la gestione e pianificazione delle risorse all'interno di un'azienda. SAP ad esempio, nato inizialmente in seno

ad IBM, produce software ERP per le aziende. Le risorse di un sistema si suddividono in: **RI** (*risorse immateriali*), **RM** (*risorse materiali*) e **RU o HR** (*risorse umane, human resources*). L'Università, gestisce solo le informazioni dello studente! Le RM sono tangibili. Le RU sono il professore, i tecnici, gli amministrativi. Questi software, o sistemi più in generale, sono molto importanti e complessi allo stesso tempo. L'acronimo CRM sta per Customer Relationship Management, a questo livello sono immagazzinati i dati degli acquisti dei clienti e si possono attuare delle tecniche MBA, ovvero di Market Basket Analysis. Nei vari sistemi superiori al CRM, per aggregare i dati in informazioni di livello superiore, si sfruttano delle potenti tecniche di clustering, Business Intelligence e Machine Learning. Con questi sistemi possiamo digerire quantità potenzialmente grandi di dati. Le informazioni possono essere di tipo analitico e di tipo sintetico. A livello CRM abbiamo ad esempio solo informazioni analitiche, più dettagliate. Tali informazioni vengono via via aggregate nei livelli superiori, mediante tecniche di cui sopra. Ci siamo serviti quindi del triangolo DMO di Anthony per capire come funzionano le organizzazioni, le persone ed i processi che elaborano le informazioni.

Un **database** è una raccolta di informazioni, il cui ciclo di vita può essere sintetizzato nell'acronimo **CRUD** (*Create, Read, Update, Delete*). Le informazioni sono straordinariamente importanti. Sapere delle informazioni ci consente di procedere operativamente in determinate direzioni piuttosto che in altre. Imparare a progettare i sistemi informativi serve a risolvere concretamente vari problemi delle persone. Abbiamo quindi a che fare con un mondo di informazioni ed organizzazioni, le quali giacciono in complesse reti. Servono tecniche di modellazione e di soluzione del problema complessivo. Un'impresa consiste in delle risorse: RI, RM, RH. Qual è il valore associato a questi oggetti? Le risorse umane, dal punto di vista dell'organizzazione, hanno un enorme valore; costituiscono il valore stesso dell'azienda, sono delle risorse non alienabili la cui sostituzione può avere effetti negativi sull'azienda stessa. Anche le RM sono delle risorse importanti, possono essere vendute per creare valore. Tuttavia, rivestono una grande importanza le risorse immateriali, ovvero le informazioni. Basti pensare a cosa accadrebbe se venissero cancellati tutti i dati dal database dell'università!!! Le informazioni all'interno delle organizzazioni hanno un valore direttamente rapportabile a quello che l'organizzazione fa. Oggi il mercato è strutturato in: Primario, Secondario, Terziario e Terziario Avanzato. Noi ingegneri informatici ci collociamo nel settore terziario. Facendo dei conti, considerando ad esempio il mercato telefonico, esso vale circa, a livello mondiale, un migliaio di miliardi di dollari. Questi soldi fanno girare un mercato enorme, ed in gioco vi sono degli interessi pazzeschi! Grandi aziende come Apple, Samsung, o Microsoft decidono del guadagno e della perdita di queste enormi moli di denaro. Oggi quando parliamo di Google o di Facebook, parliamo di tutto il mondo che c'è dietro. Trattano il settore della Comunicazione, ma lo mutano profondamente: " *Bisogna creare il bisogno*", dicono gli esperti di marketing e dinanzi a una apparente frase banale, si celano queste importanti dinamiche, che un progettista informatico deve assolutamente sapere. Non si tratta più di questioni legate all'ottimizzazione del codice ad esempio! Le informazioni sono quindi le risorse fondamentali delle organizzazioni: è una dimensione molto importante da conoscere.

Vediamo ora l'organizzazione come un Sistema Dinamico, che viene attraversato da input e produce in uscita degli output. Il direttore ha bisogno di conoscere i cosiddetti **KPI** dell'organizzazione (*Key Performance Indicator*), per sapere se l'azienda sta andando bene o meno. Queste informazioni vengono interpretate attraverso il cosiddetto **cruscotto aziendale**. Alla stregua di una macchina, le aziende si guidano e chi le guida ha necessariamente bisogno di questo cruscotto. Le informazioni devono quindi essere sempre presenti. Un'organizzazione può essere vista come una black box, che riceve in ingresso Materie Prime ed informazioni, queste vengono elaborate mediante rispettivamente sistemi operativi ed informatici, e produce alla fine in output dei prodotti o servizi, ed altre informazioni. Troviamo i cosiddetti **EIS** (*Enterprise*

Information System) a tal proposito.

1.1.3 Lo scenario della progettazione - Progetto e realizzazione

Come si colloca il concetto di modello nell'ambito dell'ingegneria? Consideriamo l'esempio dello scaldabagno: quando accendiamo il termostato la temperatura aumenta nel tempo con un andamento esponenziale fino a quando ad un certo punto non inizia a decrescere e così via. Una rappresentazione analitica di questo problema ci porterebbe intuitivamente a considerare la temperatura come una variabile rilevante mentre il colore della spia del termostato come una variabile irrilevante. Un **modello di analisi**, quindi, è una rappresentazione minimale di un sistema fisico depurato dagli effetti secondari in cui possiamo identificare una relazione I/O utile ai nostri scopi. I **modelli di sintesi** sono la concettualizzazione di un oggetto che sto creando. Il **requisito** è il modo in cui si traduce la fattibilità tecnica del sistema che si sta progettando, i requisiti costituiscono le possibili soluzioni del problema. I Goal sono i motivi per cui mi rivolgo al progettista, essi vengono risolti dai requisiti. Il progettista ha l'obbligo di rendere al cliente dopo qualche giorno dall'intervista un modello concettuale (MC), un'astrazione del **modello fisico** del problema. Il **modello concettuale** deve rendere l'idea di come il progettista ha pensato di risolvere il problema, il tutto ricorrendo ad un linguaggio comprensibile per il committente e facendo uso di illustrazioni grafiche, i Mockups. Il progettista ha a disposizione pochissimi minuti per illustrare la sua idea al committente e a convincerlo della buona riuscita del suo lavoro.

Dal momento che il committente in generale può essere incerto sulle richieste e potrebbe cambiarle in corso d'opera si redige un documento formale in cui si fissano i requisiti e si fanno firmare dal committente. Questa fase non è remunerativa pertanto è consigliabile non sprecare risorse economiche. Qualora il modello concettuale dovesse andar bene si firma un contratto tra le parti ed il committente è tenuto a dare un acconto. Il **modello logico** (ML) è un documento tecnico interpretabile da professionisti e tecnici esperti del dominio in cui sono contenute informazioni dettagliate circa lo sviluppo del progetto. Occorre definire uno standard di interfaccia tra il team e il progettista. Il team prende visione anche del modello concettuale ed infine produce il **modello fisico** (MF). Nella fase di progettazione iniziale è fondamentale predisporre anche tutti i casi di test nonché le possibili eccezioni per poi verificarle man mano che blocchi del software vengono costruiti. Il consulente è un professionista, generalmente un analista, che viene pagato dal committente per supervisionare esternamente tutto il processo di sviluppo del software e per curare i suoi interessi. Il suo compito consiste nel controllo del progettista, del team, supervisiona le release intermedie del SW e si occupa anche di collaudarlo.

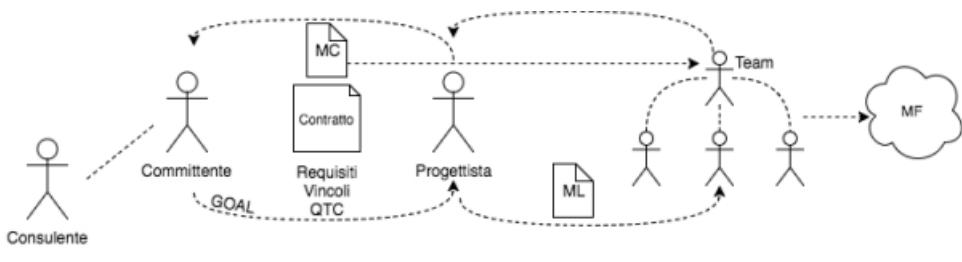


Figura 2: Lo scenario della progettazione

Figura 1.2: Scenario della Progettazione

1.1.4 Scenario Ingegneristico della Progettazione

Goal + requisiti + vincoli + test = contratto

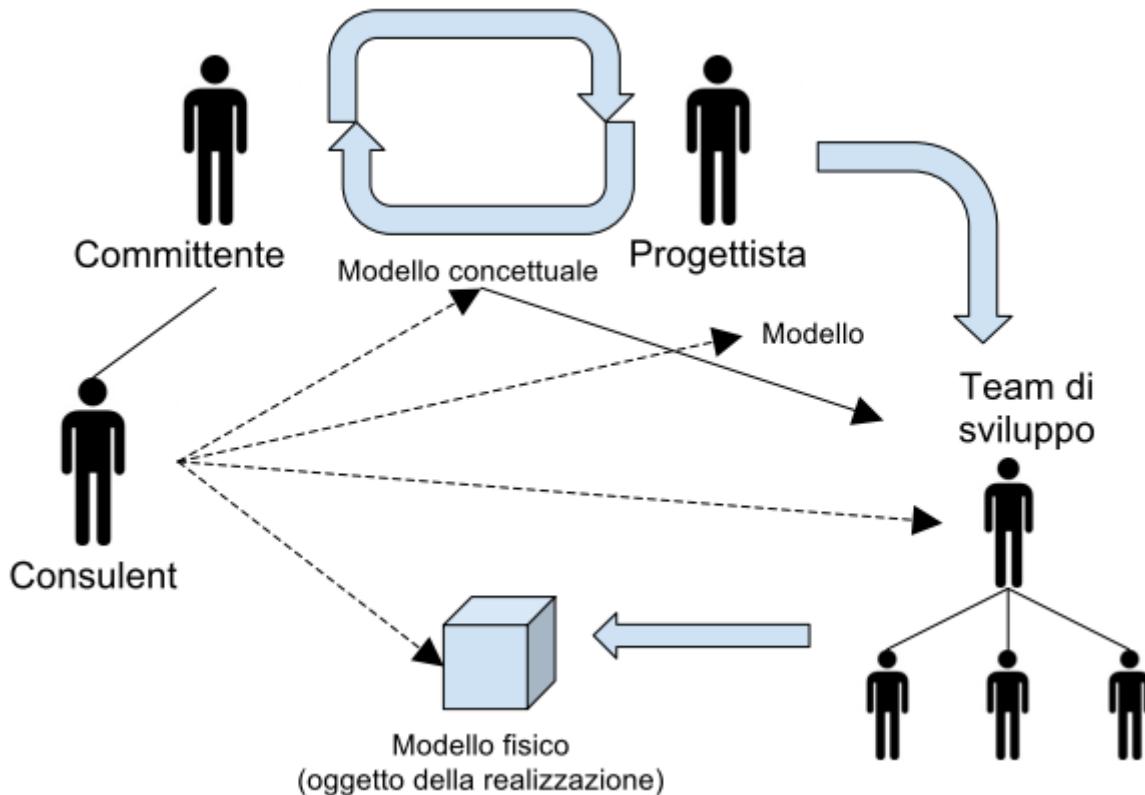


Figura 1.3: Scenario della Progettazione

Lo scenario tipico del progetto ingegneristico è il seguente: abbiamo un **committente** che richiede un **prodotto** (nel caso dell'ingegneria informatica, un'applicazione), un **progettista** che, parlando col cliente, estrae i goal del progetto, i requisiti, i vincoli di qualità, tempo e costo e i test da effettuare e dà al committente un documento, chiamato **modello concettuale** che conterrà le stime dei costi e del risultato finale del prodotto (nel caso di progetti legati al mondo dell'informatica, spesso corrispondono ai mockup) che si assocerà, alla fine, al **contratto**. Il progettista raffina il suo modello concettuale creando un altro documento che si chiama **modello logico** (che contiene i calcoli e specifiche tecniche maggiori di quelle presenti nel modello concettuale) e, insieme al modello concettuale, sarà la base da cui partiranno i **membri del team di sviluppo** (composta generalmente da un team leader e dagli sviluppatori) e si occupa di trasformare i modelli logico e concettuale, nell'**oggetto della realizzazione** (nell'ingegneria informatica corrisponde col **modello fisico**). Il committente, per assicurarsi che il lavoro sia corretto e con un buon rapporto qualità prezzo, si affida ad una terza figura, il **consulente** che, grazie alla sua esperienza pregressa, affianca il committente per verificare ogni fase del progetto. Le risorse materiali costano poco nella fase di sviluppo di un progetto dell'ingegneria informatica, il costo maggiore in un progetto lo si ha nelle risorse umane e immateriali. Avendo questa

importante informazione, questo modello ci permette di fare una stima dei costi molto precisa: si utilizzano i ***function points*** che, data una serie di specifiche, permettono di capire quanto tempo ci vuole per realizzarle. Dato il tempo di sviluppo in anni uomo, si può semplicemente moltiplicare questo valore per il costo annuo di uno sviluppatore per avere il costo totale del progetto. A questa documentazione bisogna aggiungere anche la **matrice RACI** che serve per dire “chi ha fatto una determinata parte del progetto” in modo da dare all’azienda produttrice la possibilità di tracciare la qualità di un ingegnere o di uno sviluppatore. Ricapitolando, un modello può essere fatto per il committente o per i membri del team ma, indipendentemente, è un insieme di regole e simboli noti (per esempio, in un modello dell’architettura di una casa, un segmento con un arco di circonferenza di 90 gradi indica una porta). Nel mondo dell’ingegneria informatica, i modelli statici e dinamici sono tipicamente logici, ma possono anche essere usati nel modello concettuale da mostrare al committente. Per esempio, se volessimo creare un gioco di scacchi, dovremmo creare nel diagramma delle classi una classe “casella”, una classe “scacchiera” (aggregazione di caselle), dopodiché si crea una classe generica “pezzo” che verrà implementata diversamente per ogni pezzo (pedoni, cavalli, alfieri, ecc.). Questo tipo di progettazione, dove il modello guida la progettazione, si chiama ***model driven engineering***. Si usano le tecniche di modellazione come quelle di ***forward e reverse engineering***. La prima permette, guardando il modello, di creare il software; viceversa la seconda. Ogni tipo di progettazione ha un limite. Nel caso della progettazione *model driven* si ha che il *reverse engineering* ha un costo e non è del tutto automatico, quindi ciò porta ad avere casi in cui si inizia creando il modello, si crea il codice e si modifica la struttura senza modificare il modello. Se si cambiano parti di un progetto senza modificare il modello a lungo andare si dimenticherà il motivo di tale modifica. Per questo motivo sono nati i processi di **BPR** (*business process re-engineering*). Tale processo, chiamato anche “prato verde”, permette ai nuovi team di evolversi autonomamente, senza l’obbligo di seguire metodi precedentemente utilizzati.

1.2 Architettura di un Progetto Software

1.2.1 L’hardware

Ogni calcolatore è basato sul modello della macchina di Von Neumann che presenta una parte logica (CPU), una o più memorie, uno o più dispositivi di input/output e un bus che collega il tutto.

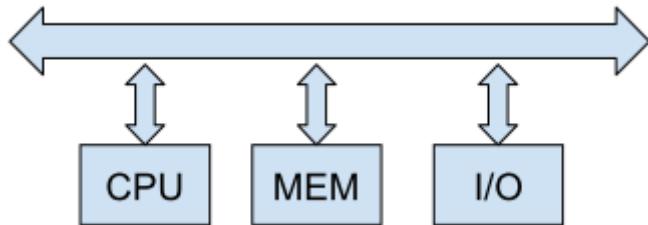


Figura 1.4: Modello della macchina di von Neumann

La CPU contiene le unità ALU (*Arithmetic and Logic Unit*), FPU (*Floating-Point Unit*), CU (*Control Unit*) e dei registri. La memoria contiene i programmi (la sequenza di istruzioni) e i dati. In questo modello, il calcolatore, al suo avvio, esegue l’istruzione alla posizione numero 0, poi la sua successiva e così via in **sequenza**. Il modello di Von Neumann permette anche di

variare il flusso di esecuzione, attraverso la comparazione e il salto. Possiamo dividere un'architettura software in diversi livelli. Il primo livello che si può trovare è il **livello hardware**, che si basa sul modello della macchina di Von Neumann. Il livello successivo è il **livello macchina** che implementa le istruzioni di esecuzione (DO), di comparazione (CMP) e di salto (JMP) nel linguaggio macchina. Questo tipo di programmazione è di tipo **imperativo sequenziale**. La sequenza può essere variata tramite CMP e JMP. Il linguaggio macchina è composto da un alfabeto di 2 caratteri (0 e 1) quindi troppo di basso livello per l'essere umano. I linguaggi che implementano queste istruzioni si chiamano **linguaggi assemblativi** (Assembly è il più famoso), nel corrispettivo livello (**livello assemblativo**). Un comando a livello assemblativo corrisponde a una serie di istruzioni a livello macchina. Il risultato di questo tipo di programmazione è anche chiamato **codice spaghetti**, dato dal grande numero di salti tra le varie righe di codice. Il **linguaggio strutturato** viene introdotto per ridurre gli errori causati dal livello assemblativo, ancora di livello troppo basso e per l'eccessiva libertà che si ha nel programmare. I linguaggi che fanno parte di questa categoria sono il C, il FORTRAN, il Pascal, e il Basic. La caratteristica di questi linguaggi è quella di avere le strutture di controllo (if, while, for, ecc...), che sono una unione di più comandi di livello assemblativo.

Il livello successivo è il **livello a oggetti**, o OODP (object oriented design and programming) che contiene i linguaggi Java, C++, Eiffel, Ruby, Python e molti altri. Questo tipo di linguaggi ha 3 particolari caratteristiche: ereditarietà, polimorfismo e encapsulamento. L'**ereditarietà** è quella caratteristica che permette il riuso e l'estensione delle classi e del codice (utilizzando le librerie di classe). Il **polimorfismo** si applica ai metodi delle classi. Permette di avere uno stesso metodo che agisce in modo diverso a seconda del tipo di classe che gli viene passato (per esempio, la somma di 1 e 2 è diversa se queste variabili sono interi o stringhe). Il principio dell'**incapsulamento** impedisce di modificare degli attributi o richiamare metodi "privati" dall'esterno della classe stessa. Questo principio permette di non avere dei *side effect* che si avrebbero invece nei linguaggi strutturati. Il problema di questo livello è che non c'è un mapping diretto tra i modelli e gli oggetti (ci sono algoritmi chiamati ORM che cercano di risolvere questo problema, ma non sempre ci riescono perfettamente).

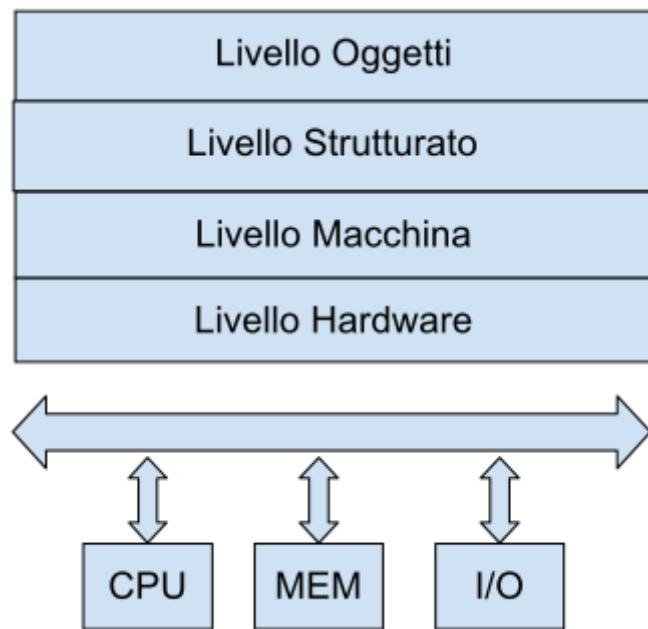


Figura 1.5: Architettura hardware-software

1.2.2 Il software - Architettura a tre livelli

Un software che utilizza una base di dati (data-centric application) può essere decomposto in tre parti: il **data layer** è il livello dove vengono salvati i dati (il database), il **business rule layer** è il livello logico del progetto, il **presentation layer** è il livello che permette di far interagire l'utente con l'applicazione (può essere un messaggio su schermo, un comando vocale, un'interfaccia a gestione, ecc...). Nella storia, il presentation layer si è evoluto con l'avanzare della tecnologia:

- quando si programmava utilizzando il linguaggio macchina, gli input erano le schede perforate e i nastri magnetici;
- con il linguaggio assemblativo sono stati introdotti i primi comandi a riga di comando (con l'introduzione dei monitor);
- con l'evoluzione a livello strutturato si è arrivati al concetto di WIMP (windows, icons, menus, pointers) che ha rivoluzionato l'utilizzo dei calcolatori;
- infine si ha l'evoluzione della WIMP in GUI (Graphic User Interface), l'introduzione delle gesti, delle interfacce vocali e delle percezioni aptiche.

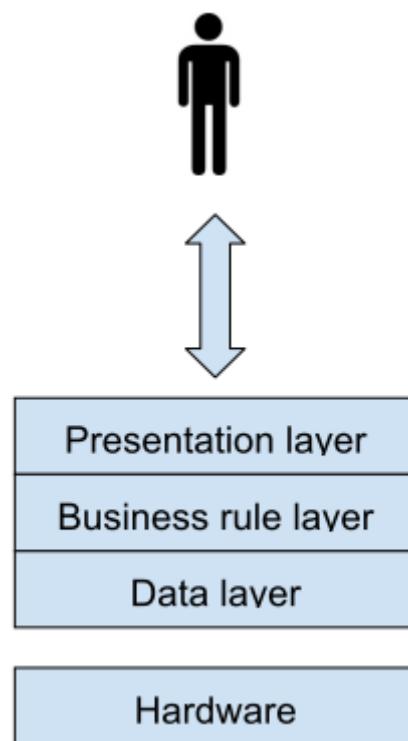


Figura 1.6: Architettura software a tre livelli

1.3 Modellazione dei dati

Argomento centrale della lezione: **MODELLAZIONE DEI DATI**. Ripartiamo brevemente da ciò che nelle scorse lezioni abbiamo chiamato scenario ingegneristico della progettazione. Impareremo a costruire tre tipi di modelli: modello concettuale, modello logico e modello fisico. Ricordiamo qual è il ruolo di ognuno di essi.

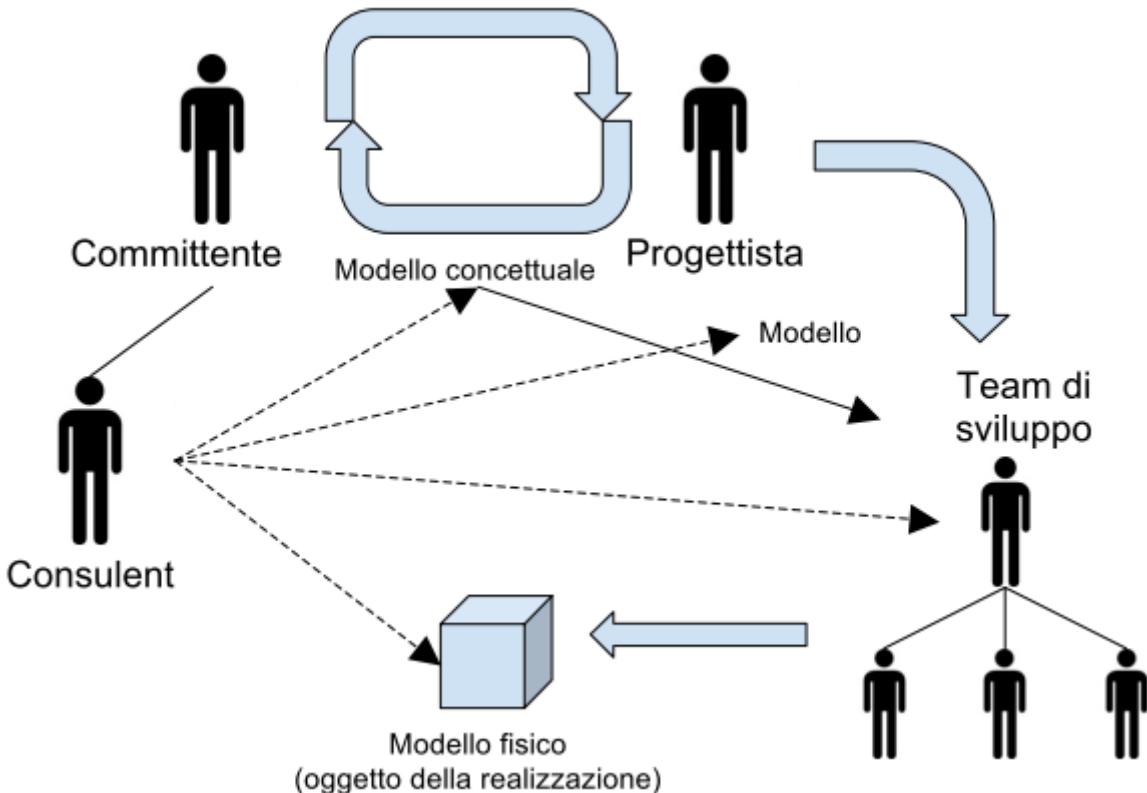


Figura 1.7: Scenario della Progettazione

Lo scenario tipico del progetto ingegneristico è il seguente: un **committente** che richiede un **prodotto** (nel caso dell'ingegneria informatica, un'applicazione), si rivolge ad un **progettista** che estrae i goal del progetto, i requisiti, i vincoli di qualità, tempo e costo e i test da effettuare. Il progettista produce per il committente un documento, chiamato **modello concettuale** che conterrà la stima dei costi e del risultato finale del prodotto (nel caso di progetti legati al mondo dell'informatica, spesso corrisponde al mockup). Il progettista raffina il modello concettuale creando un altro documento, il **modello logico** (che ha finalità più tecniche, contiene calcoli e specifiche tecniche). Modello concettuale e modello logico saranno la base da cui partiranno i membri del **team di sviluppo** (organizzato tipicamente in forma gerarchica, composta da un team leader e dagli sviluppatori) per trasformare queste informazioni nell'**oggetto della realizzazione** (nell'ingegneria informatica corrisponde con il modello fisico). Il committente, per assicurarsi che il lavoro sia corretto e con un buon rapporto qualità-prezzo, si affida ad una terza figura, il **consulente** che ha la responsabilità di supervisionare e verificare ogni fase del progetto. È importante tenere presente che il progettista deve essere abile a presentare e mettere a confronto diverse soluzioni per il medesimo problema. Ogni soluzione deve specializzare un

aspetto in particolare, perché committenti diversi potrebbero essere interessati a diversi aspetti dello stesso problema e magari vorranno porre enfasi su uno di questi. Un buon progettista deve saper dare una direzione alle idee, impostare una strategia di risoluzione. In qualità di ingegneri, dobbiamo essere in grado di creare modelli, di progettare una soluzione sia dal punto di vista visuale, che tecnico, che quantitativo. Per disegnare e particolareggiare ogni aspetto della modellazione ci si avvale di un insieme di strumenti. Un esempio nel caso di modellazione di software è UML, che è una raccolta di diversi tipi di diagrammi, ognuno dei quali aiuta il progettista a descrivere un particolare aspetto di ciò che deve realizzare. Per progettare sono quindi necessarie diverse abilità, perché bisogna modellare diversi aspetti. Nell'ambito dell'ICT distinguiamo tre aspetti principali:

- Hardware (ex. processor, display, elementi di networking, ecc.);
- Software (ex. sistema operativo, software applicativo, driver, ecc.);
- Dati: Un progetto non è ben progettato se non vengono considerati tutti e tre gli aspetti sopra elencati.

Un progetto non è ben progettato se non vengono considerati tutti e tre gli aspetti sopra elencati.

1.3.1 ESEMPIO – APPLICAZIONE CLIENT-SERVER

Nella figura sottostante è rappresentata la struttura hardware di un'applicazione Client-Server: c'è un client connesso a Internet attraverso un ISP e un dispositivo chiamato Google Server. Queste due macchine sono collegate dal punto di vista hardware da una rete, basata su tecnologie come TCP/IP. Quello hardware non è l'unico livello da considerare, bisogna occuparsi anche della parte software. Infatti un dispositivo avrà bisogno di un sistema operativo e di applicazioni.

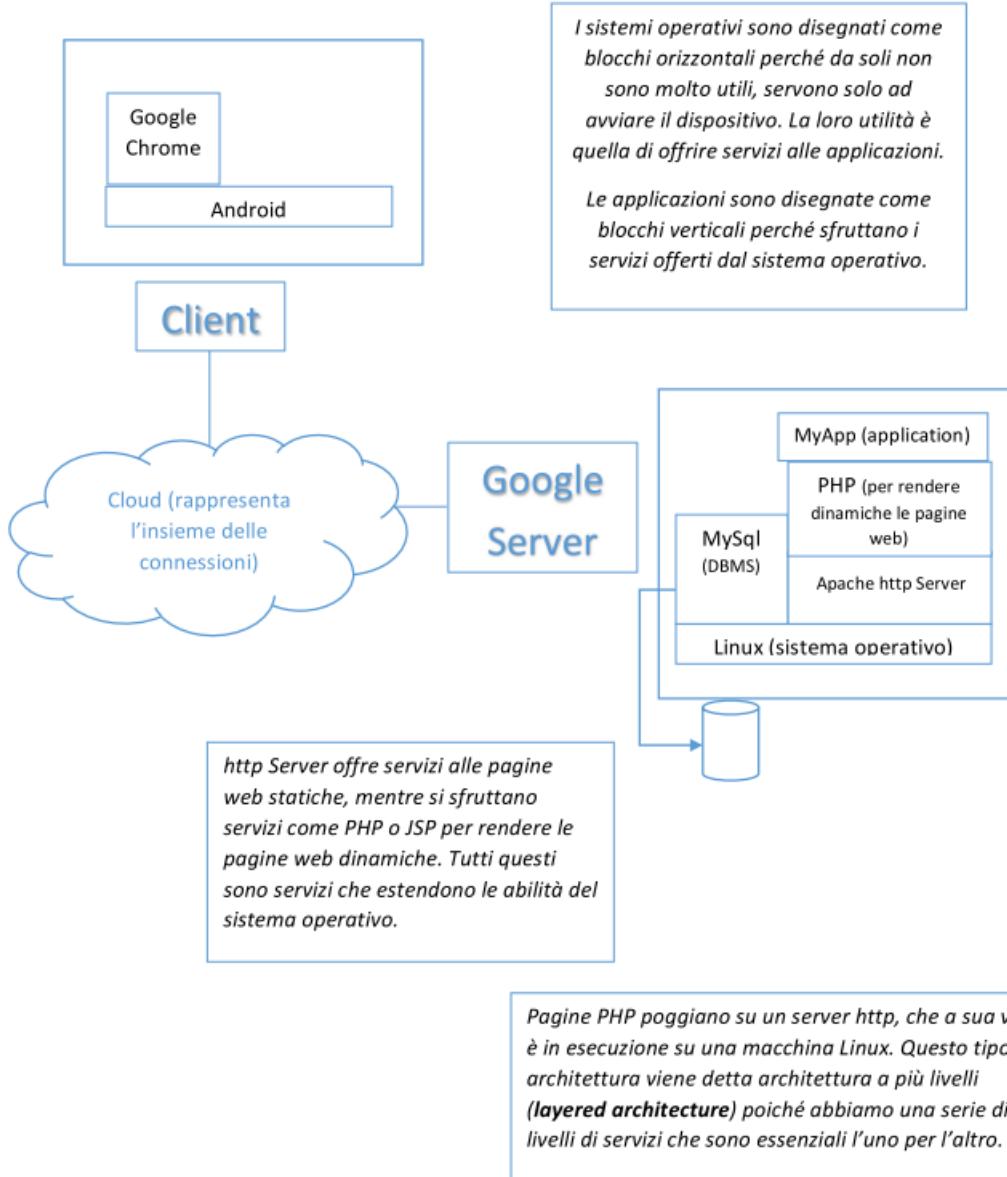


Figura 1.8: Scenario Google Client-Server

Il Server Hardware è una macchina destinata a soddisfare richieste. Quando si disegna l’architettura software si progetta il tipo di software da inserire nella macchina hardware e le richieste che potranno essere realizzate. Infine, bisogna progettare anche un’architettura dati. Oggi disegneremo la nostra prima Data Architecture. Come fare? In generale quando parliamo di Software Engineering, le architetture sono disegnate in termini di classi. Ogni classe è specificata da un nome, degli attributi e dei metodi. Le classi possono interagire tra loro ed esistono diversi tipi di relazioni tra di esse: una classe può utilizzare un’altra classe, aggregarla, specializzarla, ecc. Il problema sorge quando bisogna mappare queste classi in Data Structures per gestire la persistenza e la serializzazione dei dati. Si dice che un’applicazione è in grado di gestire la persistenza dei dati se questi sopravvivono a diverse sessioni di utilizzo dell’applicazione. Dare alle classi la possibilità di realizzare la persistenza dei dati non è sufficiente, poiché si ha anche la necessità di lavorare con essi. Normalmente facciamo utilizzo di database non solo per memorizzare dati, ma anche per recuperarli in un ordine differente da quello utilizzato per memorizzarli. Abbiamo bisogno di eseguire diverse operazioni sui dati: crearli, cancellarli, modificarli, trasformarli, ecc. Per fare questo utilizziamo delle funzioni di alto livello, come le Associative Functions. Riscrivere queste funzioni ogni volta che scriviamo un programma è molto dispendioso e poco pratico. Perciò si utilizza un **DBMS (DataBase Management System)** come MySql, un software orientato al data processing, che rappresenta i dati in maniera differente. Un DBMS organizza in “librerie” le funzioni di cui abbiamo bisogno per l’elaborazione dei dati. Utilizzare le funzioni di alto livello fornite da un DBMS rende il nostro codice più efficiente e la programmazione più veloce e meno costosa. Quindi non è necessario che le nostre classi gestiscano direttamente la persistenza e implementino funzioni di ricerca, perché abbiamo a disposizione del software specifico che si occupa della gestione dei dati. Non scriveremo codice in termini di Object Oriented code, ma faremo quello che è chiamato **ORM (Object Relational Mapping)**. Un esempio di Object Relational Mapper è Hibernate. Per esempio, possiamo scrivere un programma Java che richiede dati ad un database relazionale. Un ORM come Hibernate si occupa di trasformare le richieste delle classi Java in richieste al database, traducendo le operazioni sugli Oggetti in operazioni su Tabelle del database. La progettazione di un’applicazione richiede la realizzazione di questi passi:

- Goal;
- Requisiti;
- Architettura Sw, Architettura Hw, Architettura dati;
- Test e deployment;
- Operation

1.3.2 ESEMPIO

Iniziamo a vedere come modellare concettualmente i dati. Data una situazione come quella descritta dall’Entity Relationship Diagram in figura, un ingegnere deve essere in grado di immaginare come sarà il software.

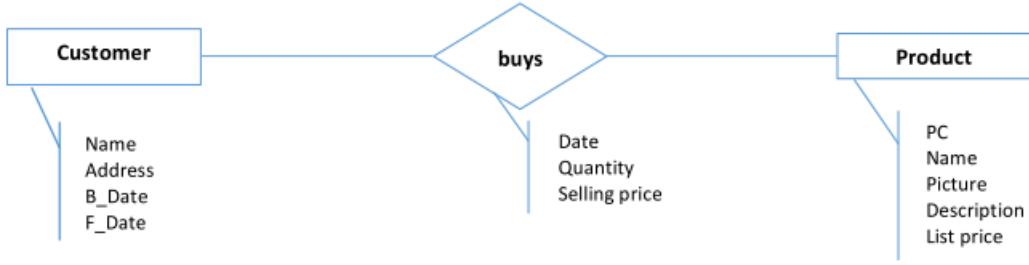


Figura 1.9: Entity Relationship Diagram

Customer e **Product** prendono il nome di entità, poiché possono esistere anche da sole, sono definite senza dipendere dagli altri elementi del database. **Buys** è una relazione e non può esistere da sola, dal momento che non avrebbe alcun significato se posta fuori da un contesto (esempio: non posso definire un acquisto senza indicare l'acquirente e il prodotto acquistato). Possiamo rappresentare un database come un grafo, in cui ogni **Entità** è un nodo e ogni **Relazione** è un arco. Su questo grafo possiamo individuare diversi percorsi, che corrispondono a interrogazioni al database (query).

1.3.3 ESEMPIO - THE COMPANY DATABASE

Facciamo un passo in avanti analizzando qualcosa di più complesso. Consideriamo l'esempio *The Company Database* del libro di testo.

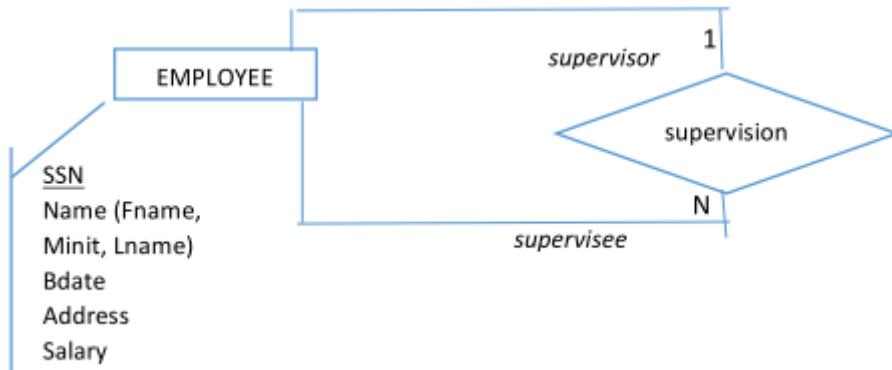


Figura 1.10: The Company Database

Notiamo che:

- L'attributo SSN dell'entità employee è sottolineato. Con la sottolineatura si indica la **chiave primaria**, ovvero l'attributo (o l'insieme di attributi) che identifica univocamente un'istanza di una certa entità. Non potranno esistere due employees aventi attributo SSN con lo stesso valore;
- I numeri indicano la **cardinalità** della relazione. Nell'esempio abbiamo che un employee può supervisionare N employees ed un employee può essere supervisionato da un solo employee (relazione uno-a-molti).

Esandiamo lo schema precedente, aggiungendo altre entità...

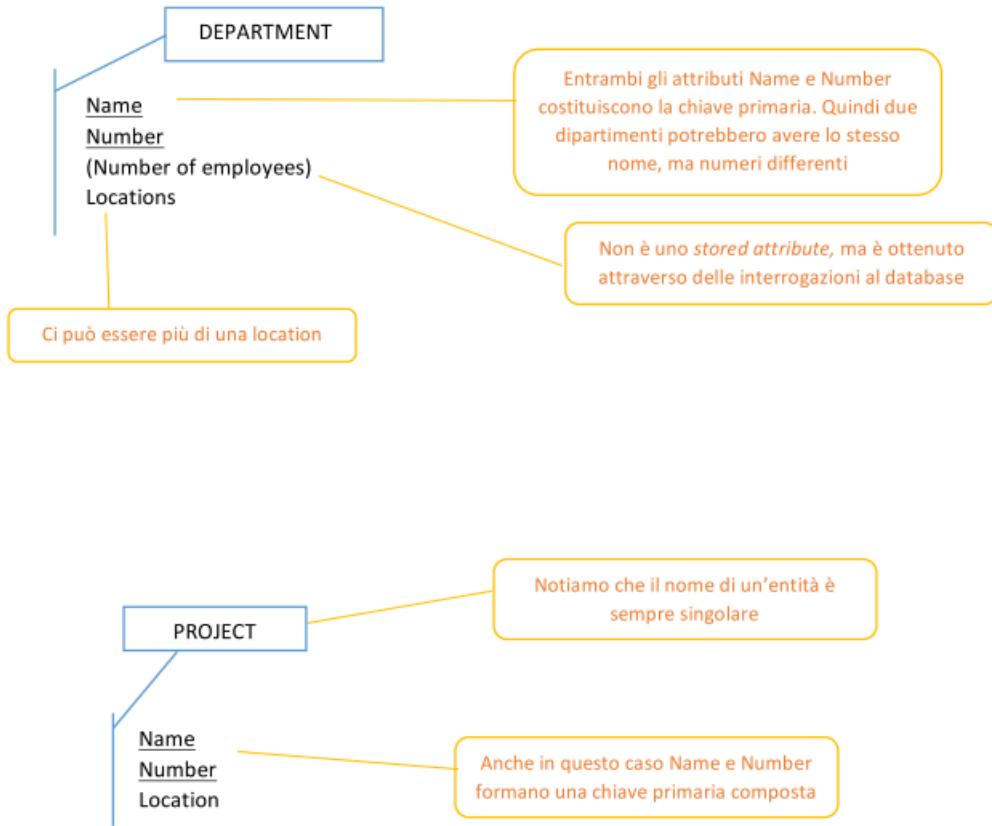


Figura 1.11: The Company Database

... e le relazioni che intercorrono tra di esse:

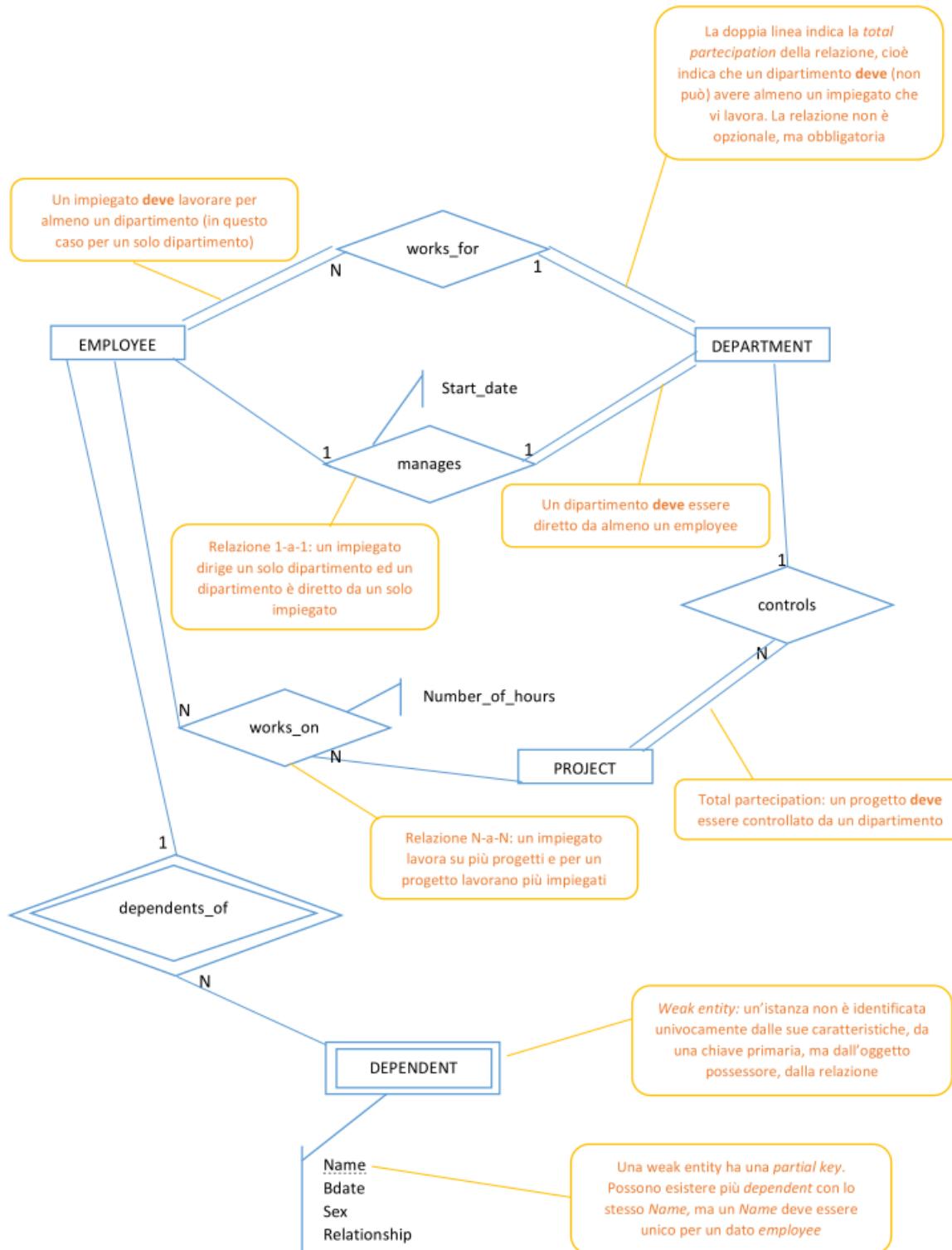


Figura 1.12: The Company Database 2

Floriana Accoto

1.4 Conceptual Model

Nella lezione precedente abbiamo visto come realizzare il data modeling, composto da CM (Conceptual model), LM (Logical model), FM (Physical model). Gli elementi principali del CM sono tre: Entity type, Relation type, Attributes.



Figura 1.13: Elementi CM

Combinando questi elementi si crea il CM. Naturalmente esistono delle regole e dei vincoli da rispettare, in quanto si sta parlando di un linguaggio formale. Ad esempio non sono consentite operazioni del genere:

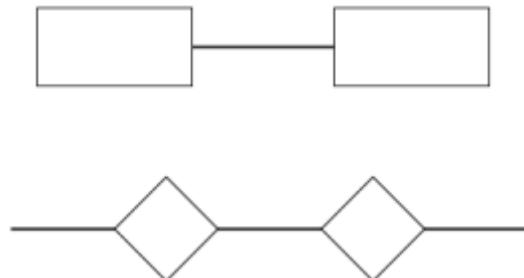


Figura 1.14: Operazioni proibite in un ER

vale a dire non è possibile collegare direttamente due entità o due relazioni.
La logica da seguire è la seguente,

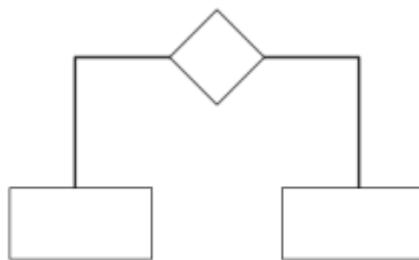


Figura 1.15: ER corretto

tenendo conto che le entità devono avere necessariamente degli attributi, mentre le relazioni possono anche non averne.

Il modello concettuale fornisce anche i seguenti vincoli:

- Cardinalità;
- Total Participation;
- Primary Key.

Per ogni entità partecipante ad una relazione viene specificata una cardinalità di relazione. Essa è una coppia di numeri naturali che specifica il numero minimo e massimo di istanze di relazione a cui un'istanza dell'entità può partecipare. Esistono tre tipi di cardinalità:

- 1 a 1;
- 1 a molti;
- Molti a molti.

Nell'ultimo caso conviene mettere due lettere differenti perché riportare la stessa lettera (ad esempio n to n) fa capire che le entità possiedono lo stesso numero di partecipanti, ma questo è errato. Infatti non si è in grado di prevedere a priori il numero esatto dei partecipanti, portando il progettista a fornire un'informazione errata del modello.

Vediamo degli esempi, per comprendere meglio come differenziare i vari tipi di cardinalità:



Figura 1.16: Relazione 1 a 1 → A (person) - R(has) – B (Driving license)

La patente è un'entità che esiste anche se non ha un possessore, così come la persona, ed ha come attributi numero, ente, data di rilascio, ecc. In questo caso ha senso parlare di totally participation solo nel caso della patente, che non può non essere associata ad una persona. Mentre la persona non deve necessariamente avere una patente e potrebbe avere come attributi il nome, cognome, codice fiscale. N.B. Un esempio errato è quello di definire come entità il codice fiscale di una persona. Infatti ogni individuo possiede un codice fiscale che lo caratterizza, ma il codice fiscale è un'attributo dell'entità persona, in quanto è una caratteristica della persona. Inoltre possiamo affermare che il codice fiscale è un autonomous entity type.

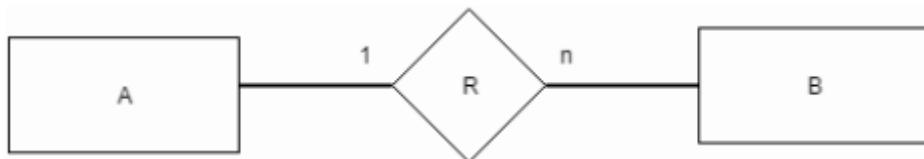


Figura 1.17: Relazione 1 a molti → A (Person) – R (Own) – B (Car)

• È una relazione uno a molti poiché una persona può possedere più macchine, al contrario una macchina non può essere posseduta da più persone. Di conseguenza, in questo caso, non si può parlare di total participation, perché una persona può non avere un'auto e un'auto può non avere un proprietario.



Figura 1.18: Relazione molti a molti → A (Car) – R (Has) – B (Optional)

• Questo tipo di relazione ci permette di introdurre il concetto di pattern, di cui si parla nell'appendice.

1.4.1 Il valore NULL

Il valore NULL è un valore speciale in un DB, infatti la PK (chiave primaria), cioè un singolo raggruppamento di attributi dell'entità che ci permette di identificare univocamente un'istanza dell'entità stessa, deve essere necessariamente di tipo NOT NULL, vediamo il perché. È buona norma evitare di inserire valori di tipo NULL all'interno del DB, in quanto sono indice di mancanza di informazioni e al momento dell'estrazione dei dati, non si potrà comprendere se il dato non è stato inserito, se non se ne conosce il valore, se c'è stato un errore in fase di inserimento ecc. Pertanto la presenza di valori di tipo NULL significa che non solo non si conosce il valore di un determinato attributo, ma non si sa neanche il motivo per cui quel valore non è presente. La maniera corretta di inserire in un DB devi valori nulli è quella di inserire delle stringhe vuote. Ad esempio se si vuole inserire in un record una persona che non ha un secondo nome, mentre tra gli attributi della persona compare, il modo corretto di procedere è quello di inserire una stringa vuota, cioè “”. ESEMPIO: Se si volesse calcolare l'età media di un gruppo di persone inserite in un DB, è sufficiente che ci sia un unico campo in cui compaia un valore di tipo NULL per fare non essere in grado di calcolarla. In quel caso si potrà avere solo una stima dell'età media, ma non l'età media esatta.

$$3 + 2 + \text{NULL} = \text{NULL}$$

1.4.2 ANALIST



Figura 1.19: Customer & Designer

1.4.3 SCHEMA

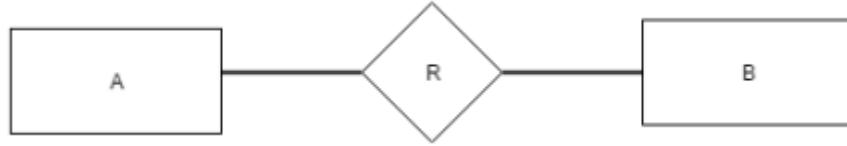


Figura 1.20: Schema ER

È importante capire che gli schemi di database usati nei precedenti esempi forniscono una INTENTIONAL DESCRIPTION OF DATABASE, cioè, mediante l'uso di un linguaggio formale si è in grado di fornire una descrizione logica del database. Si useranno ora nuovi elementi che portano a fornire una EXTENSIONAL DESCRIPTION OF DATABASE. Rientrano in questa categoria:

- Tabelle;
- Entity Set.

Le tabelle in un database relazionale sono un insieme di elementi che sfruttano un modello in cui ci sono colonne verticali (ogni colonna è identificata con il nome di un attributo) e righe orizzontali, dove la cella del database rappresenta il punto in cui colonna e riga si intersecano. Ogni tabella del database coincide con l'entità presa in analisi, dove le colonne rappresentano gli attributi. Se la quantità degli attributi ha un valore fissato, le righe invece hanno un valore variabile in quanto rappresentano il numero di istanze contenute nell'entità. La riga di una tabella prende il nome di record della tabella. Le tabelle sono alla base delle relazioni. Un altro tipo di EXTENSIONAL DESCRIPTION OF DATABASE non si basa sull'uso delle tabelle ma sulla SET THEORY. Per poter fornire la definizione di entity set, bisogna definire che cos'è un entity type. Un entity type definisce una collezione (o set) di entità che hanno gli stessi attributi. Ogni entity type è descritto dal nome e dai suoi attributi. La collezione di tutte le entità di un particolare entity type nel database in qualsiasi punto nel tempo è chiamato entity set o entity collection.

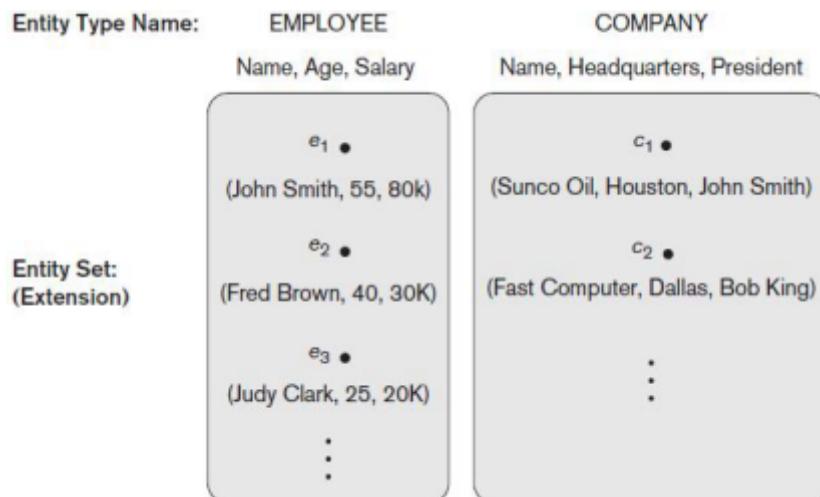


Figura 1.21: Entity Set

L'obiettivo che ci si pone è quello di usare uno stesso DB per soddisfare più bisogni. ES. Una catena di negozi, che ha sedi diverse, utilizza copie dello stesso DB per registrare gli acquisti degli utenti. Naturalmente viene usata una copia dello stesso DB in ogni sede e ovviamente gli acquisti registrati saranno differenti in ogni sede.

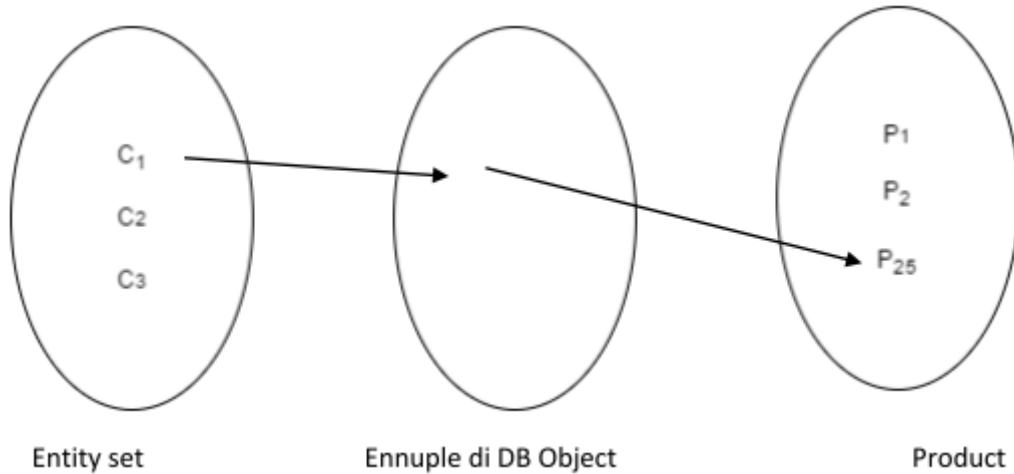


Figura 1.22: Entity Set Eulero-Venn

La precedente descrizione del modello è stata estratta dal libro “Fundamentals of database systems”, mentre a lezione l'esempio si basava sulle entità Venditore e Prodotto. Il legame tra un'istanza di product e una di customer mi permette di ottenere una relazione che prende il nome di Tuple o ennupla. La tupla è una lista finita ordinata di elementi. Gli elementi di una tupla sono contenuti all'interno di “()”, ed ogni elemento è separata da una ”,”. Si comprende facilmente che gli elementi di una tupla sono gli attributi dell'entità presa in considerazione. Quindi, una relazione è un set di tuple degli oggetti del database. Gli attributi che hanno il ruolo di caratterizzare le entità e le relazioni possono essere definiti concettualmente:

- semplici;
- composti;
- multipli;
- multipli & composti.

Un attributo è semplice quando fornisce una singola informazione di base che caratterizza l'entità, assumendo un singolo valore (professione di una persona). Un attributo è composto quando è creato tramite l'uso di attributi semplici (l'indirizzo è composto dal tipo di via, nome della via, indirizzo civico, paese). Un attributo si dice multiplo quando a esso possono essere associati più valori dello stesso tipo contemporaneamente (locazione nel caso di un dipartimento universitario). Un esempio di attributo multiplo e composto è il numero di telefono, il quale è composto dal prefisso e dal numero. Durante la creazione di un database, tutti gli attributi composti e multipli devono essere convertiti in attributi semplici perché nei database relazionali sono accettati solo gli attributi semplici.

1.4.4 Come iniziare la progettazione di un DB

Si parte con carta e penna, realizzando uno schema, non necessariamente corretto, in base alle richieste del committente. Solitamente non si riesce a creare un DB corretto subito. Vediamo un esempio. Il modello in figura evolve dinamicamente per ogni soluzione migliore che si riesce a trovare. Si è notato, in questo caso, che era più opportuno non considerare il nome del manager e la data di inizio del suo lavoro come attributo dell'entità department, ma creare una nuova entità di tipo impiegato che è in relazione con department.

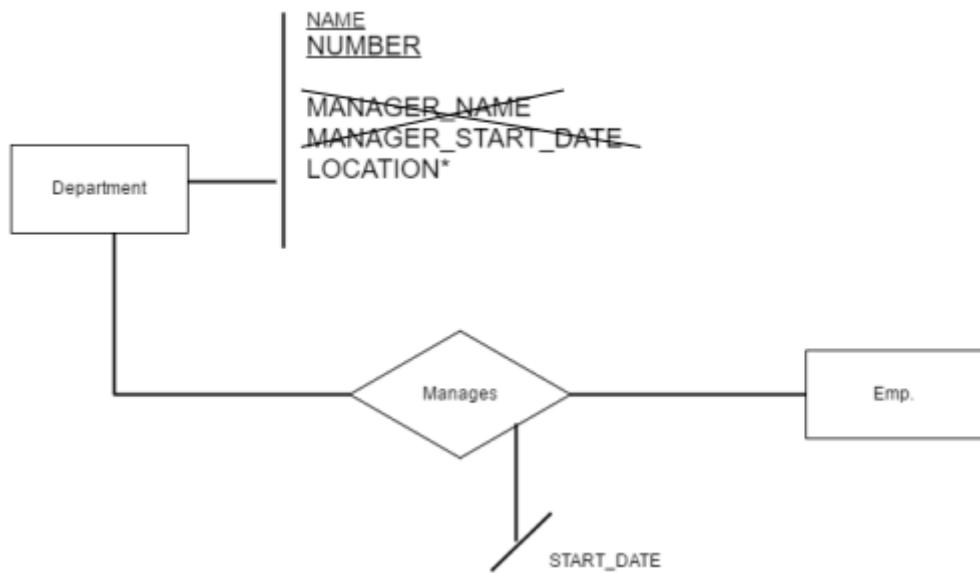


Figura 1.23: Incremental ER

1.4.5 Ternary relation type

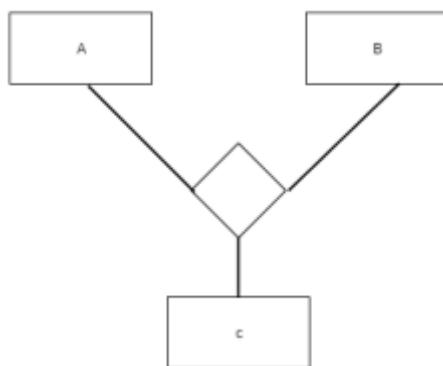


Figura 1.24: Ternary Relationship

Nel caso in cui si ha la presenza di tre entità, in funzione della relazione usata, siamo in grado di estrarre delle informazioni più precise. Per farlo si sfrutta una relazione ternaria,

che diversamente da quella binaria, mette in collegamento attraverso una singola relazione tre entità. Le informazioni ottenute da una relazione ternaria contengono informazioni più precise rispetto a quando vengono usate delle relazioni binarie. Possiamo esprimere la precedente affermazione tramite disuguaglianza matematica:

$$\text{TernaryRelationship} > \text{BinaryRelationship}$$

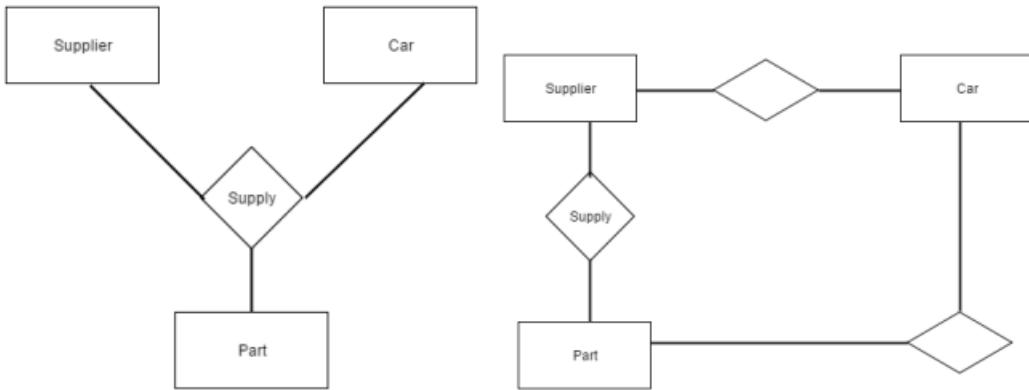


Figura 1.25: Ternary Relationship and Three Binary Relationship

Gli schemi rappresentati in figura sono equivalenti. Solo che, come già detto, 3 relazioni binarie contengono meno informazioni di una relazione ternaria. Per mantenere la stessa quantità di informazioni si fa uso della REIFICAZIONE, che consiste nel trasformare una relazione in un'entità. In questo modo si ha la presenza delle tre entità, un'entità reificata e tre relazioni binarie che collegano le entità all'entità reificata. Quindi continuo ad usare relazioni binarie anziché utilizzare relazioni ternarie, senza perdere informazione. In questo modo l'informazione estrapolata dalla relazione ternaria e l'informazione estrapolata dalla procedura di reificazione è equivalente:

$$\text{TernaryRelationship} := 3 \text{ BinaryRelationship} + 1 \text{ EntityType}$$

L'uso di questa procedura è a discrezione esclusivamente del progettista.

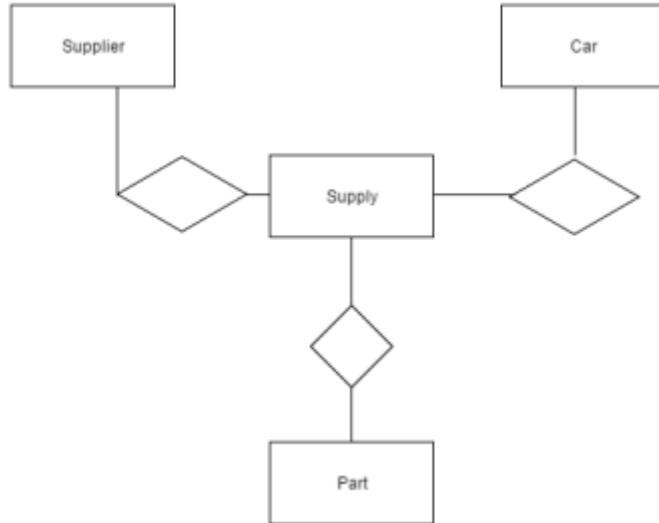


Figura 1.26: Reification of a Ternary Relationship

1.5 APPENDICE

1.5.1 DESIGN PATTERN

Anche per i db esistono dei design pattern.

- **X toX (es. Object type of object)** → Consiste nella distinzione tra oggetti unici, cioè rappresentati da un codice identificativo (automobile, armi, ...), e oggetti che sono istanze dello stesso tipo assolutamente identiche (capi di abbigliamento, generi alimentari, ...). Quindi la differenza consiste nel fatto che, si può acquistare, ad esempio, un'istanza di un oggetto (pacco di pasta) e non il singolo oggetto oppure il singolo oggetto se identificabile univocamente con qualche criterio. Questo pattern è utile in quanto è un tipico errore quello di inserire, all'interno del db, un oggetto al posto del tipo di oggetto.
- **Temporal Dimension** → Prima di dire lo scopo dell'utilizzo di questo pattern, è di fondamentale importanza rimarcare un concetto da tenere a mente quando si ha a che fare con il mondo dei DB, cioè che i dati sono fondamentali, sono importanti, sono una risorsa preziosa e a meno di casi eccezionali non vanno eliminati dal DB perché potrebbero ritornare utili in qualsiasi momento. Si sta facendo riferimento al concetto di Data Mining. ES. Se si dovessero perdere i dati (risorse immateriali) di una DB di una banca, che è un ente composto da risorse umane, materiali e immateriali, comporterebbe la perdita di tutte le informazioni relative ai correntisti o ai titoli bancari, di conseguenza le risorse materiali non avrebbero un possessore. Detto ciò, passiamo ad un esempio pratico per comprendere l'utilizzo di questo pattern. Come si può vedere nella figura sottostante, il pattern Temporal Dimension esprime il fatto che alcuni attributi possano variare nel tempo, pertanto alle volte si ritiene opportuno avere uno storico delle informazioni, nel nostro caso variazioni di dati riguardanti il proprietario o l'automobile. In altre parole, sto aggiungendo la dimensione temporale al mio DB, per avere traccia delle varie versioni di un dato oggetto.

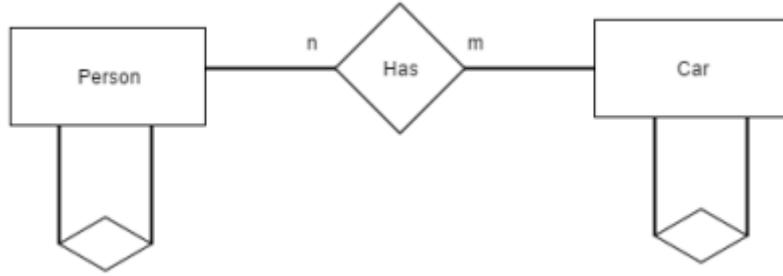


Figura 1.27: Temporal Dimension

Cristian Annicchiarico
 Mattia Marzano
 06/10/2016

1.6 Esercizio - Fermate Autobus

I diagrammi Entità-Relazione (ER) sono stati estesi e migliorati con lo scopo di includere tecniche di modellazione tipiche del design object-oriented. Il risultato è una nuova classe di diagrammi definiti EER (Extended Entity-Relationship), che si avvicinano di più al concetto di diagramma delle classi UML.

Esercizio: Siamo una compagnia di bus e vogliamo un'architettura del DB che ci aiuti a gestire il nostro business. Dobbiamo prendere in considerazione i seguenti punti:

- Bus;
- Linee (fermate, collegamenti);
- Impiegati;
- Viaggiatori;
- Biglietti (settimanali, mensili, annuali).

Gli aspetti precedenti sono ciò che chiamiamo **requisiti**, che il nostro progetto deve soddisfare. Alcune assunzioni aggiuntive verranno considerate nel prosieguo dell'esercizio in modo da ridurre la complessità del progetto. Senza perdita di generalità, considereremo solamente biglietti che sono validi in un lasso di tempo maggiore di 1 giorno (ovvero, settimanali/mensili/annuali), poiché i biglietti giornalieri non ci permettono di avere dettagli su chi stia usando quel biglietto, mentre, ad esempio, i biglietti settimanali richiedono una sorta di tessera che contiene i dati personali del viaggiatore.

Importante: dobbiamo sempre ricordare che vogliamo un'architettura dei dati che soddisfi le esigenze di tutti i nostri stakeholder. In altre parole, il nostro design pattern è: Un database, multiple applicazioni (ovvero, interfacce e presentazioni differenti per gli stessi dati: vedi Figura 1). Anche se il database è lo stesso, utenti diversi usufruiscono di viste differenti degli stessi dati. Per spiegarlo attraverso un esempio, prendete un qualsiasi gioco di carte da tavolo: il database è “tutte le carte sul tavolo”, mentre ogni utente percepisce una diversa presentazione (infatti, lui o lei può solo sapere quali carte scoperte sono poste sul tavolo, quali carte sono nella sua mano, e quali carte erano nella sua mano).

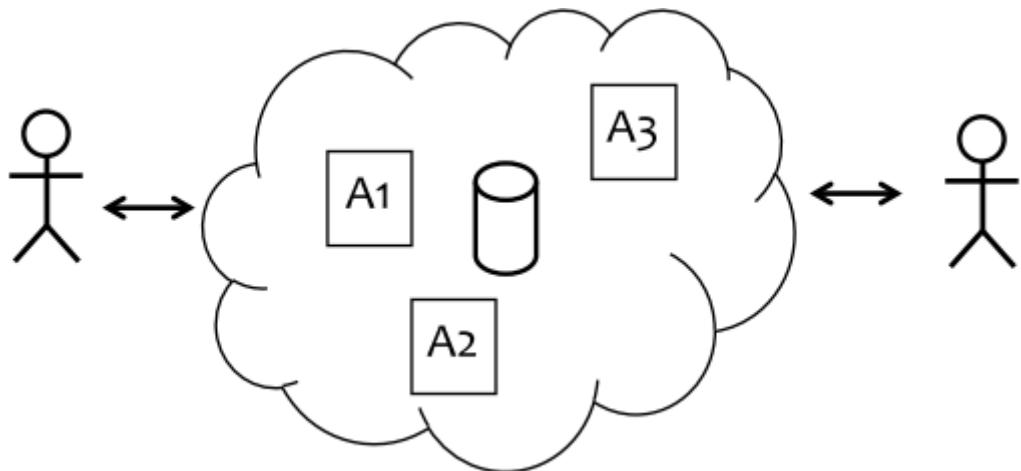


Figura 1.28: Un database, multiple applicazioni

Al fine di svolgere l'esercizio, consideriamo separatamente I requisiti. Prima di tutto, consideriamo le entità “Traveller” ed “Employee”. Al momento, modelliamole entrambe come un'unica entità “Person” con gli attributi appropriati:



Figura 1.29: Entità ”Person”

Consideriamo ora linee con fermate e collegamenti. Questo ci porta alla creazione dell'entità “Line” (Figura 3).

- **Domanda:** dovremmo modellare le fermate di una certa linea come un attributo multiplo?
- **Risposta:** No, poiché se consideriamo una fermata interposta tra due linee, dovremmo dichiararla due volte nel nostro database (ovvero, come un attributo sia della prima che della seconda linea). Questo violerebbe la terza regola dei database.

Terza regola dei database: Mai aggiungere ridondanza al vostro database!

Anticipazione: In un diagramma E-R, la chiave primaria è solo concettuale (dobbiamo scegliere un campo che sia unico). Nella pratica, quando traduciamo un diagramma E-R nel suo corrispondente modello relazionale (modello logico), aggiungeremo un campo ID, responsabile dell'unicità di ciascuna entry nelle nostre tabelle. Esso sarà la nostra chiave primaria, ma non è visibile agli utenti.



Figura 3: Entità "Line"

Figura 1.30: Entità "Line"

Come specificare quali fermate attraversa una linea? Prima di tutto, dobbiamo modellare le fermate. Lo facciamo tramite un'entità "Stop".

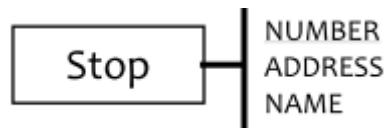


Figura 1.31: Entità "Line"

Come modellare i collegamenti tra le fermate? Abbiamo considerato varie alternative. Esaminiamole in ordine.

- **Prima alternativa per modellare le fermate**

Poiché una linea è una sequenza di fermate, può essere considerata come una coda. Nei database, una coda può essere modellata creando una relazione ricorsiva (Figura 5). Comunque, questo approccio ha un problema. Prendete come esempio la Figura 6, in cui consideriamo due linee (rossa e verde). Nella linea rossa, la fermata blu è la numero 2, mentre nella linea verde, la fermata blu è la numero 3. Dovremmo modellare la fermata blu come la seconda o la terza fermata della coda? La risposta è: nessuna delle due, poiché la fermata blu deve comportarsi come terza fermata della coda per la linea rossa e come seconda fermata della coda per la linea verde. Con una simile architettura del database, come possiamo tenere traccia di quale fermata è situata a un certo punto di una certa linea?

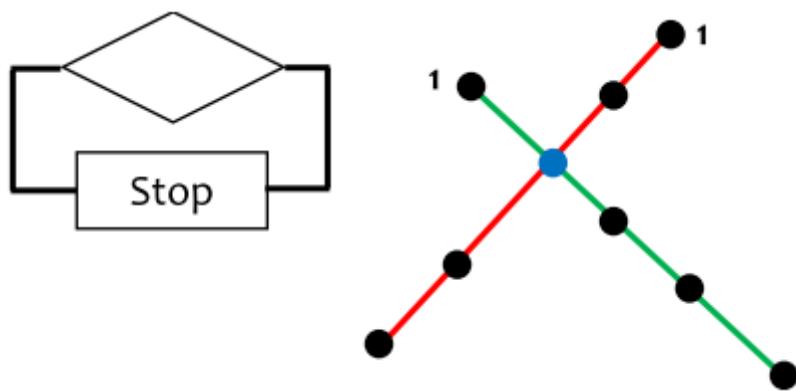


Figura 1.32: A sx: Modellare una coda (prima alternativa), dx: Due linee che condividono la stessa fermata

- **Seconda alternativa per modellare le fermate**

Poiché il problema con la precedente soluzione era la mancanza di un collegamento tra la coda di fermate e le linee, un’idea sarebbe rimodellare lo scenario come mostrato in figura successiva. Comunque, questa architettura evidenzia due problemi:

- E’ troppo complessa. Un requisito banale (una coda) non dovrebbe creare una grande complessità nel diagramma;
- La relazione 1-1 tra “Queue” e “Line”, insieme con la totale partecipazione su entrambi i lati della relazione 1-1, suggerisce che di fatto “Queue” e “Line” siano lo stesso oggetto. Quindi, potremmo comprimerle in un’unica entità.

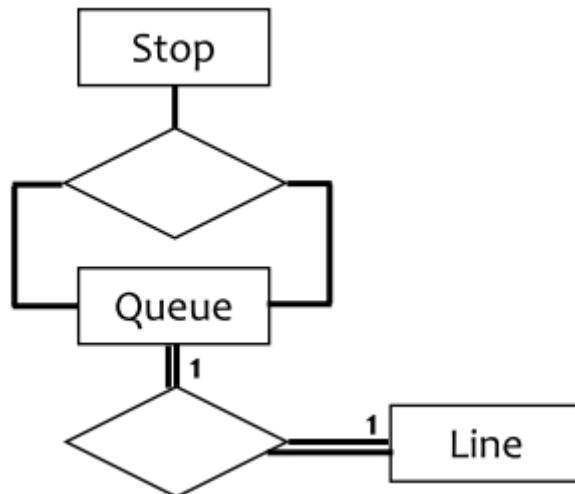


Figura 1.33: Modellare una coda (seconda alternativa)

- **Terza alternativa per modellare le fermate (la migliore)**

Dopo tali considerazioni, abbiamo finito per semplificare la nostra architettura. La migliore alternativa (finora) è avere una semplice relazione N a M tra “Line” e “Stop” con un attributo #Step che consente la creazione di una lista ordinata. E’ stato aggiunto un attributo addizionale Minutes, per modellare il tempo necessario, a partire dall’inizio di una corsa, per attraversare quella fermata. Inoltre, è un attributo multiplo, che dunque consente varie tempistiche per diverse ore della giornata. Lo scenario è riassunto nella successiva figura. Questa soluzione ha vari benefici:

- La complessità della ridefinizione della lista è spostata dentro all’applicazione. Quando una fermata è rimossa da una linea, deve essere eseguito da parte dell’applicazione il controllo di coerenza della lista;
- Ha il giusto livello di complessità in paragone ai requisiti specificati;
- Consente a un utente interessato di conoscere in che momento una linea attraversa una fermata: è sufficiente aggiungere il numero di minuti a partire dalla partenza del bus.



Figura 1.34: Modellare una coda (terza alternativa)

Consideriamo ora il requisito dei biglietti. Non andremo a modellarlo come un’entità poiché significherebbe dire che i biglietti mensili esistono a prescindere dalla persona che li usa o a prescindere della linea per cui il biglietto è stato erogato. Questo ovviamente non è ciò che accade nella realtà. Motivo per cui modelleremo un biglietto come una relazione tra le entità interessate, come si può vedere dalla successiva figura. Questo è di fatto il modo più naturale di aggiungerlo nel nostro diagramma: un biglietto è un contratto, un pezzo di carta in cui sono scritte sia la persona che lo utilizza che la linea per cui può essere usato.

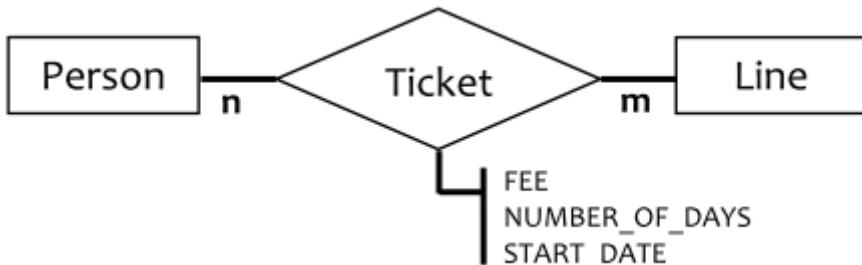


Figura 1.35: La relazione ”Ticket”

Infine, aggiungiamo il restante requisito al nostro diagramma: i bus. Prima di tutto, creiamo una nuova entità “Bus” (Prossima Figura). Abbiamo aggiunto alcuni attributi basilari, ma possiamo aggiungerne altri del tipo: numero di posti, se ha o meno supporto per persone disabili, se ha alcune utili agevolazioni per gli autisti, ecc..

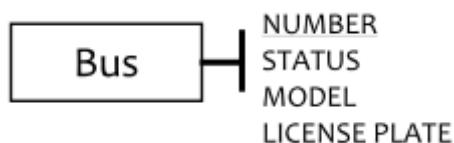


Figura 1.36: Entità ”Bus”

Ora dobbiamo analizzare due aspetti:

- Come modellare la tabella oraria per le linee?

- Come tener traccia di quale bus ha servito una certa linea in un certo giorno a una certa ora?

Per rispondere alla prima domanda, abbiamo introdotto l'entità “Race” (Successiva figura). Questa nuova entità debole modella una particolare istanza di una linea (con una tabella oraria). Mentre “Line” indica un percorso teorico attraverso alcune fermate, “Race” è una particolare istanza di “Line” che ha un orario di partenza formale per ogni giorno della settimana.

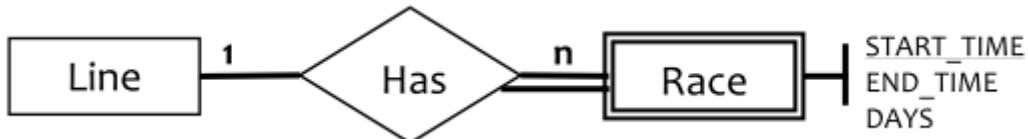


Figura 1.37: Entità “Race” e suo collegamento all’entità “Line”

Con questa nuova entità, la seconda domanda può essere riformulata nel seguente modo: come tener traccia di quale bus ha servito una certa corsa? Questo può essere banalmente ottenuto aggiungendo una relazione tra “Race” e “Bus” (Successiva figura). Il diagramma ER finale è riportato in Figura 13. Comunque, questa impostazione non considera chi ha guidato in una certa corsa. Non abbiamo esplorato questo punto nella lezione, ma può essere ottenuto in vari modi:

- Modificando la relazione “Served” in successiva figura in una relazione ternaria, in cui il terzo collegamento è attaccato all’entità “Person”;
- Aggiungendo una relazione tra “Person” e “Race” (ma questo obbliga gli impiegati a guidare sempre nella stessa corsa e non permette modifiche negli assegnamenti).

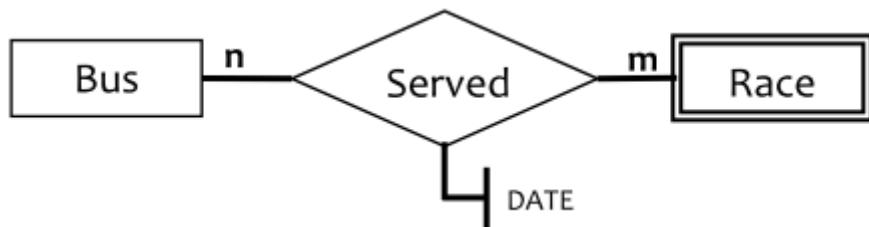


Figura 1.38: La relazione “Served”

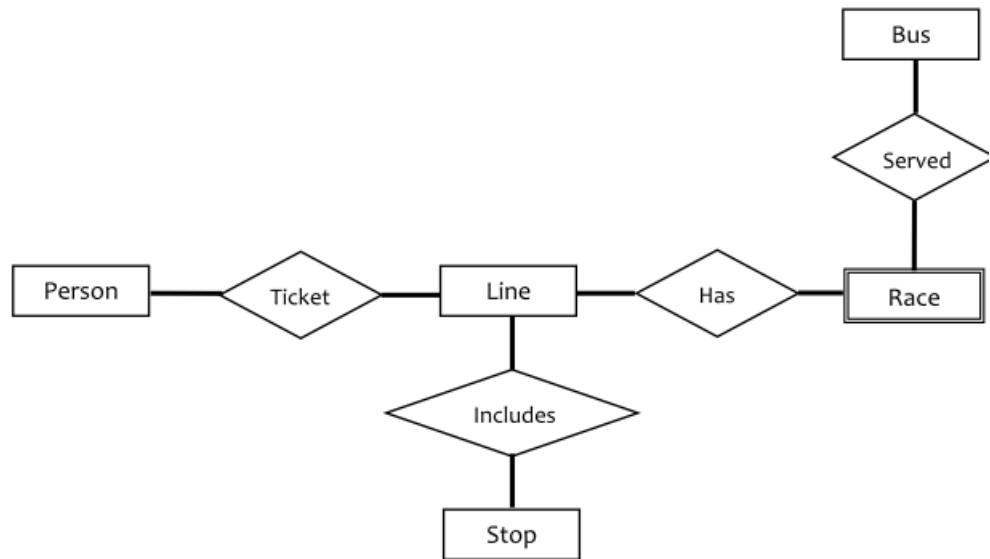


Figura 1.39: Diagramma ER finale per l'esercizio sulla compagnia di bus

Le fermate dovrebbero essere collegate alle corse? Introduciamo un altro design pattern che risolve questo problema: il Preventivo Consuntivo. Questo ci consentirà di salvare l'informazione circa il tempo effettivo in cui una corsa ha attraversato una certa fermata in un certo giorno (permettendoci, ad esempio, di ricalcolare periodicamente il tempo di arrivo stimato sulla base delle statistiche).

1.6.1 Design Pattern: Preventivo-Consuntivo

Questo pattern gestisce la previsione di un evento e la sua effettiva realizzazione. Introduciamolo tramite un esempio: la prenotazione in un hotel. La previsione è la prenotazione di una camera, mentre la realizzazione è ciò che accade effettivamente (ovvero, se la camera viene occupata o meno al tempo previsto). In questo caso, possono accadere tre situazioni:

- Qualcuno prenota la camera e la usa (prenotazione + uso);
- Qualcuno prenota la camera ma non la usa (prenotazione + non uso);
- Nessuno prenota la camera ma qualcuno la usa (no prenotazione + uso).

Di fatto possono accadere più situazioni (per esempio, vi potrebbero essere multiple prenotazioni e cancellazioni, o una prenotazione per N persone e $N \neq M$ persone che la usano, ecc...). Se il diagramma originario contiene una relazione tra due entità A e B (ad esempio "Person" e "Room") che è di tipo Preventivo-Consuntivo (ad esempio "Books" e "Uses"), questo pattern richiede di dividere la relazione in due relazioni: prevenire (Books) e consumare (Uses). Questa situazione è illustrata nella successiva figura:

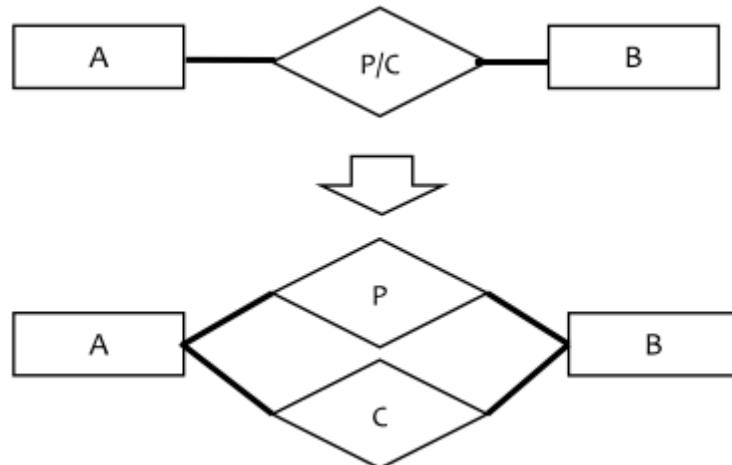


Figura 1.40: Divisione della relazione. P e C stanno rispettivamente per Prevenire e Consumare

Nell'esempio di prenotazione di un hotel, con tale architettura è possibile avere informazioni separate circa la prenotazione di camera e l'utilizzo di camera. Comunque, c'è ancora un tassello mancante: la relazione tra Preventivo e Consumazione (ovvero, quale consumazione ha seguito quale preventivo). Nondimeno, nel nostro esempio ciò non è un problema, poiché può esistere solo un utilizzo alla volta per una specifica camera: la relazione tra P e C può essere facilmente ricostruita. Se è richiesta una specifica relazione tra P e C, la si può ottenere ridefinendo le relazioni P e C (Successiva figura). Questa struttura è più forte ma davvero complessa. Nel

nostro esempio di prenotazione, ciò ci consentirebbe di conoscere quale prenotazione è stata seguita da un effettivo utilizzo. In altre parole, ci permetterebbe di sapere che una certa prenotazione è quella che ha di fatto consentito a una persona di entrare e utilizzare la camera prenotata. Consumazione = un'attuazione di un'azione preventiva.

1.6.2 Conclusione

Abbiamo specificato molti requisiti. Ciascuno di essi ha avuto impatto sulla soluzione finale. Dovremmo controllare costantemente se essi sono soddisfatti e coperti dalla nostra soluzione. Inoltre, dobbiamo sempre essere pronti a suggerire altre soluzioni quando sono richieste. L'Informazione è un'ecosistema di Produttori, Consumatori e Trasformatori.

N.B.: Non dobbiamo ottimizzare il nostro database per le query, ma per gli stakeholder (deve soddisfare appropriatamente ciascuno di essi).

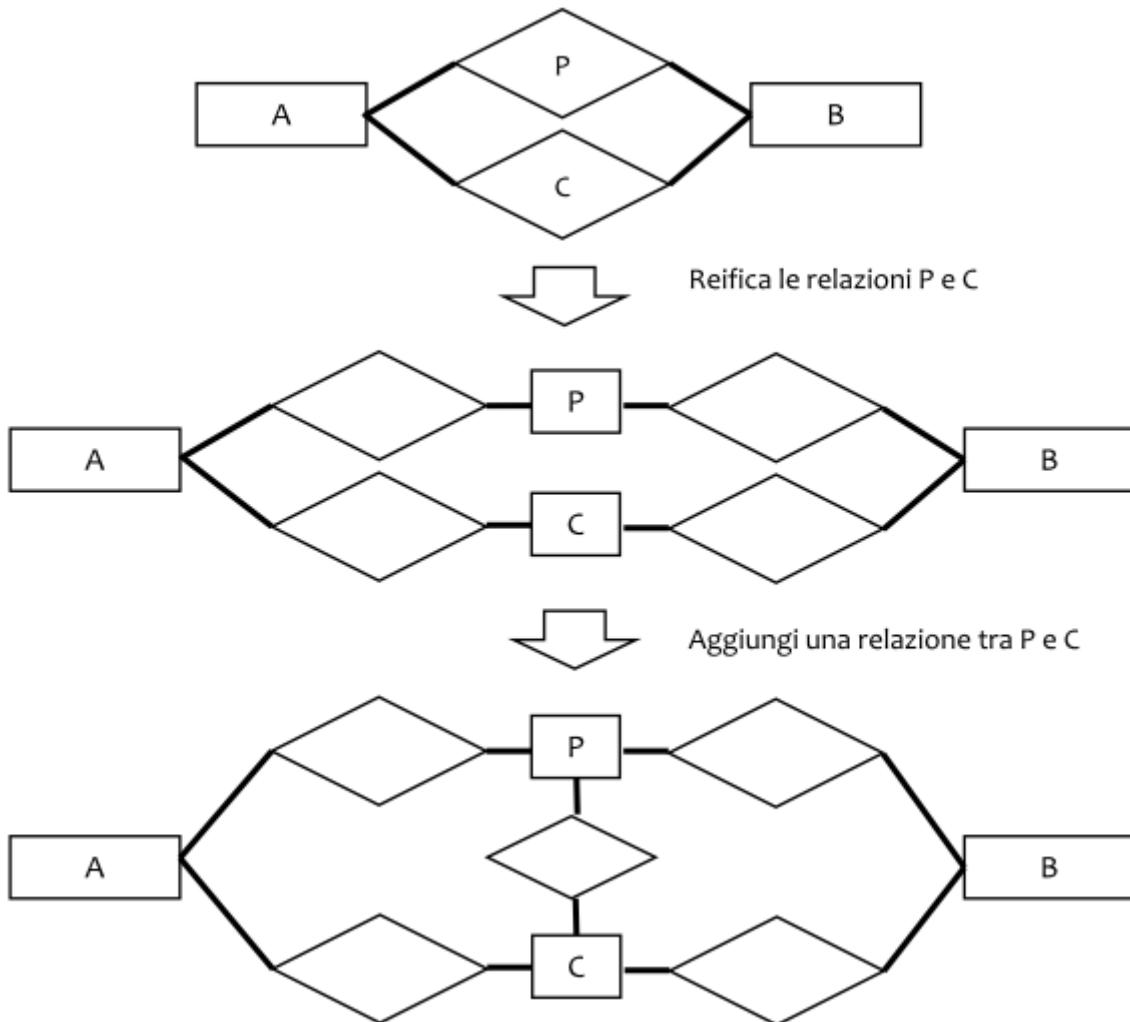


Figura 1.41: Divisione della relazione. P e C stanno rispettivamente per Prevenire e Consumare

1.7 DATA CENTRIC APPLICATION

La lezione di oggi ha lo scopo di analizzare il ciclo di creazione e modellazione di applicazioni che si occupano di database. Per l'ingegneria delle applicazioni Data Centric si utilizza un approccio differente:

MAPPING ALGORITHM Trasforma i diagrammi E.R. in tabelle relazionali:

- CONCEPTUAL MODEL (ER DIAGRAM);
- LOGICAL MODEL – RELATIONAL MODEL (Concetti Astratti);
- PHYSICAL MODEL (REAL DATABASE – MySQL).

Un database è un insieme di relazioni R_i e vincoli C_j :

$$DB = \{R_i, C_j\}$$

Le relazioni sono le tabelle, i vincoli sono le regole che gestiscono le tabelle. Ci sono dei livelli per la costruzione di un database:

- **PRESENTATION LAYER – P.L.;**
- **BUSINESS RULE LAYER – B.R.L.;**
- **DATABASE LAYER – D.L.;**

1.7.1 USER TYPES

Gli User Types sono gli agenti che estraggono le informazioni da Database in modo differente utilizzando il medesimo software. Per ognuno di essi bisogna progettare quindi diverse interfacce e diverse regole di accesso al DB:

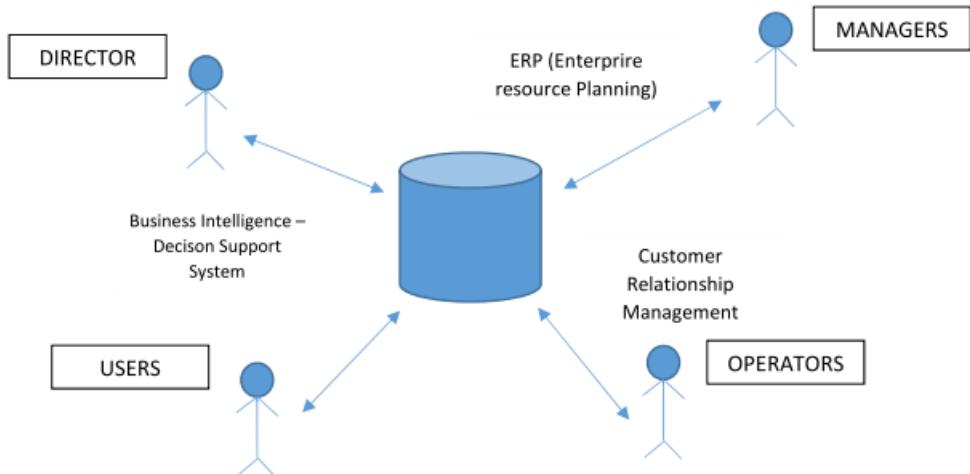


Figura 1.42: User Types

1.7.2 ESEMPIO

Usiamo un'applicazione esempio per analizzare lo schema appena spiegato (DL, BRL, PL):



Figura 1.43: Customer Buys Product Type

In questo esempio gli UserTypes sono i Customers e i Sellers e sono per lo più statici, cioè non cambiano. Mentre i Product Type sono più dinamici in quanto rappresentano le transazioni economiche.



Figura 1.44: Customer Buys Product Type - Dinamicità

← La dinamicità della relazione è molto più alta rispetto alle altre due entità. Come facciamo a estrarre differenti info per le diverse tipologie di users? Si userà una forma semplificata del **Mapping Algorithm**.

1.7.3 SIMPLIFIED MAPPING ALGORITHM

- **Ogni Entity Type diventa una tabella e i loro attributi diventano i nomi delle colonne.** Customer, products diventano tabelle, con i relativi attributi. Ogni customer diventerà un record della tabella Customer:

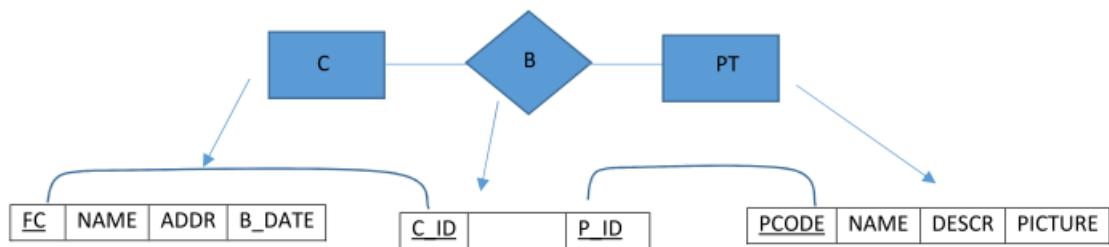


Figura 1.45: Customer Buys Product Type - Mapping

- **Ogni Relation Types diventa una tabella.** E' sempre vero in un solo caso, ovvero quando la relazione è “molti a molti” (n:m):

- Collegare la chiave esterna (F.K.) a ogni entità;
- Completare la tabella con tutti gli attributi presenti della Relation Type.
- Nel caso **n:m** si crea quello che si chiama **Relational Schema of DB**;
- Nel caso della relazione **1:n**, bisogna includere la tabella della relazione nella tabella finale dell'entità “n”

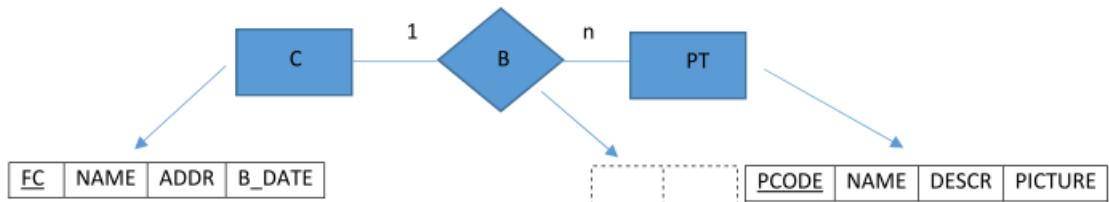


Figura 1.46: Customer Buys Product Type - 1:n Mapping

- Nel caso della relazione **1:1** si può includere la tabella della relazione in una delle due entità oppure creare una sola tabella:

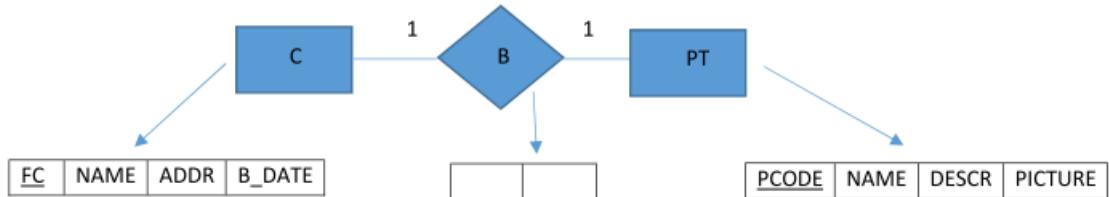


Figura 1.47: Customer Buys Product Type - 1:1 Mapping

Se cancello un record da una tabella entità, verrà cancellato ogni record relativo presente nella tabella relazione. Il discorso non vale tra tabelle entità diverse, poiché indipendenti tra loro. Riferendoci all'esempio precedente se ad esempio viene cancellato un customer, ogni transazione riferita a lui viene cancellata, lo stesso vale per i prodotti.

1.7.4 PRESENTATION LAYER FOR CUSTOMER OR DIRECTOR

Avremo differenti livelli di presentazione, uno per il customer e uno per il direttore del negozio. Per il Customer la Web Application si vedrà come:

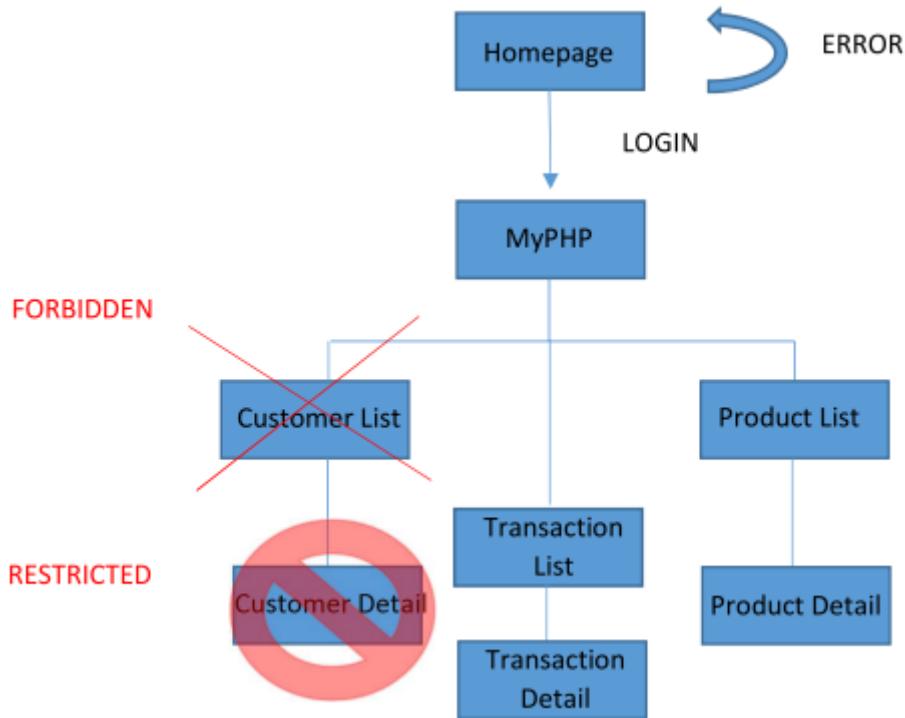


Figura 1.48: Navigation Model of a Website For Customer

Un'altra soluzione potrebbe essere:

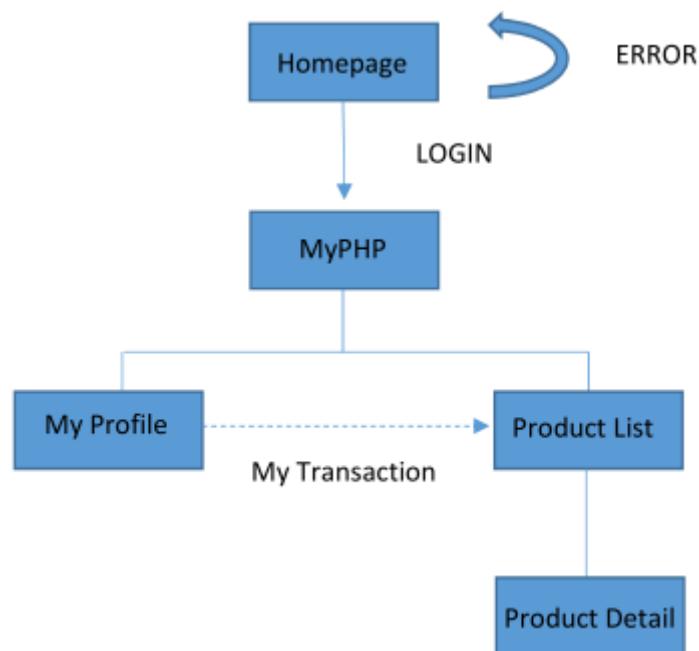


Figura 1.49: Navigation Model of a Website For Customer - Another solution

Per il Director invece, la Web Application si vedrà come:

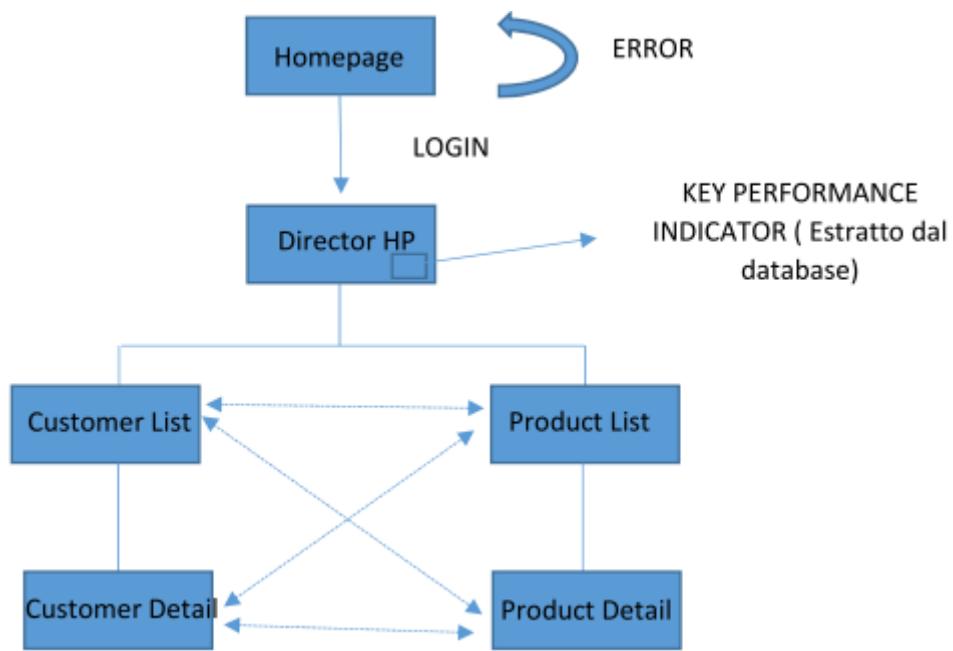


Figura 1.50: Navigation Model of a Website For Director

Il direttore ha la possibilità di creare o cancellare i prodotti e di modificare gli attributi di essi. Inoltre può ottenere informazioni sui Customers e quindi potrà poi effettuare delle scelte strategiche per migliorare le vendite con pubblicità o sconti mirati.

1.7.5 BUSINESS RULE LAYER

Man mano che si sale nella piramide abbiamo bisogno di aggregare le informazioni fino ad avere delle info sempre più riassuntive.

Quando si realizza un progetto c'è bisogno di definire:

- CONTEXT & PROBLEM DEFINITION (2 PAGINE);
 - GOAL & STAKEHOLDERS;
 - REQUIREMENTS & CONSTRAINTS;
 - DATA MODEL:
 - ER DIAGRAM;
 - RELATIONAL MODEL;
 - PHYSICAL MODEL.
 - SYSTEM MODELS;
 - SOFTWARE MODELS (MODEL VIEW CONTROLLER);

- STATE MODEL;
- PRESENTATION MODEL:
 - * IN THE LARGE NAVIGATION (Modello per mostrare le pagine e le connessioni fra di esse);
 - * IN THE SMALL NAVIGATION (Modello per analizzare ogni aspetto singolarmente);
- TEST MODEL;
- DEPLOYMENT STRATEGY.

1.8 LIST VIEW and DETAIL VIEW

Questi due concetti provengono dai sistemi operativi (WINDOWS, MACOS, XWINDOWS/UNIX) e nascono dall'idea comune detta WIMP INTERFACE:

- WINDOWS = Diversi spazi di lavoro per diverse operazioni (MULTITASKING);
- ICONS = Oggetti Interattivi;
- MENUS = Metodi associati ai propri oggetti;
- POINTERS = Selezionare un oggetto specifico.

Da questa metafora nasce l'idea di LIST VIEW e DETAIL VIEW. Se ad esempio apro il Desktop ottengo una lista di oggetti, se invece interagisco con uno di essi ottengo informazioni dettagliate.

• LIST VIEW

List View può essere una tabella nella quale sono presenti dei metodi well-known come:

- DELETE OBJECT;
- SORT;
- (MOVE);
- COPY (DUPLICATE OBJECT);
- INSERT EMPTY OBJECT;
- FILTER;
- RENAME;
- SEARCH (FIND).

Altri esempi di List View possono essere una Cartina Geografica, il Desktop o una cartella, un Sets o un Tree.

• DETAIL VIEW

I metodi well-known della Detail View possono essere:

- CHANGE (UPDATE);
- CREATE;

- DELETE;
- READ.

Angelo Cotardo
 Francesco Filieri
 12/10/2016

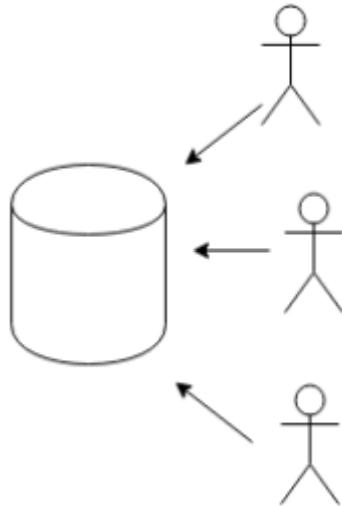


Figura 1.51: Database and User Types

Continuiamo con una discussione più dettagliata riguardo i design patterns. Nella scorsa lezione abbiamo esplorato l'intera catena che inizia dal modello e termina con le applicazioni. Il database è un luogo in cui differenti tipi di utente possono svolgere molteplici azioni, ognuno di essi è collegato a qualche processo, descritto tramite un linguaggio grafico chiamato BPML (Business Process Modeling Language) che descrive i processi in un'organizzazione. Parleremo di BUSINESS MODEL, molto importante per i nostri scopi, di DATA MODEL, usato per il database e infine di PRESENTATION MODEL e SYSTEM MODEL. Quando parliamo di CONCEPTUAL MODEL, ci riferiamo a tutti i modelli che sono molto usati per raccogliere requisiti e informazioni riguardo a ciò che è necessario fare e per trasformare poi questi in software. Approfondiremo gli aspetti di data modeling. Abbiamo già introdotto il concetto di ER che andremo ad espandere con i design patterns e con il concetto di EER (ENHANCED ENTITY RELATIONSHIP). Aggiungeremo poi anche qualche concetto riguardo la modellazione di oggetti (ereditarietà ecc.).

1.8.1 DATA PATTERN

Introduciamo il concetto di DATA PATTERN

Consideriamo il seguente schema:

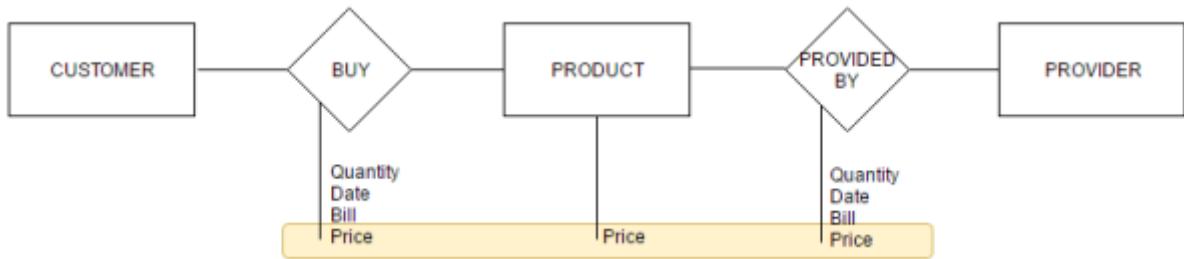


Figura 1.52: Customer Buys Product Provided By Provider

Stiamo considerando lo scenario in cui un consumatore compra dei prodotti, i quali sono forniti da più fornitori. Osservando gli attributi delle relazioni e delle entità descritte, oltre alla quantità, al tipo di prodotto acquistato e alla data di acquisto, si è definito l'attributo “price” che compare per ben tre volte come attributo delle due relazioni e dell'entità PRODUCT.

Il significato che assume tale attributo cambia:

- *Selling cost*;
- *Catalog cost*;
- *Payed cost*.

Nel primo caso esprime il costo del prodotto appena comprato, nel secondo si riferisce al prezzo specificato nel catalogo dei prodotti (non necessariamente selling price deve essere uguale al catalog price) e nel terzo caso ci si riferisce al prezzo pagato al fornitore riguardo quel prodotto, il quale ovviamente è diverso dal prezzo specificato nel catalogo. Il pattern che si è applicato in questo caso (cioè trovare lo stesso attributo in entità o relazioni differenti) prende il nome di PRICE CHAIN (o più in generale ATTRIBUTE CHAIN). Un altro tipo di pattern che presentiamo è il PRESENTATION PATTERN che non si focalizza sui dati, bensì sulla loro rappresentazione: l'ARCHIVE PATTERN. È collegato al concetto che quando memorizziamo delle informazioni, esse possono diventare col tempo poco usate.

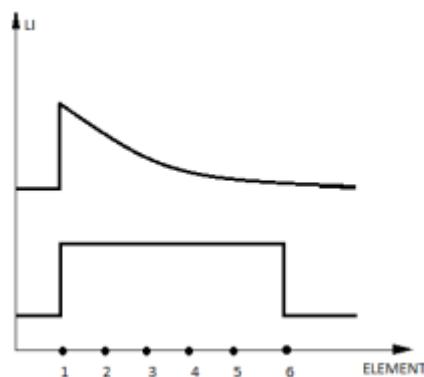


Figura 1.53: Interest Level for Objects

Dallo schema si può notare i vari livelli di interesse (LI) per ogni elemento inserito in una list view (cioè le informazioni contenute in tali elementi). In alcuni sistemi quindi, più di un elemento può avere lo stesso livello di interesse, in altri invece, come quelli usati da Google per le ricerche, le informazioni in cima alla lista avranno un livello di interesse elevato rispetto ai successivi elementi. Per gestire ciò, viene assegnato ad ogni query un ranking, cioè un indicatore in base al quale ordinare i risultati. Quindi le “risposte” con informazioni più recenti alle “domande” effettuate avranno un ranking più elevato rispetto alle altre. Tali indicatori possono essere assegnati ad esempio sulla base del numero di persone interessate ad un determinato argomento. Un altro esempio del genere, non più basato sul concetto di ranking, può essere quello di una email software, in cui le email più vecchie sono meno importanti delle ultime ricevute, pertanto ci si basa sul tempo. Un esempio in cui tutti gli elementi possono avere lo stesso livello di interesse può essere quello dei contatti Whatsapp, in cui ad ognuno di essi viene data la stessa importanza:

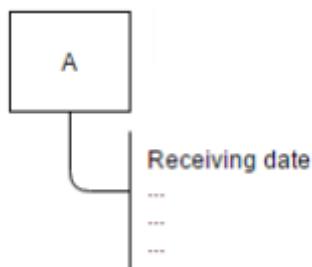


Figura 1.54: Whatsapp: Receiving Date attribute

Possiamo aggiungere a tale entity type un **ranking attribute** (o altri tipi di attributi che collegano il livello di attenzione ai dati). In questo caso consideriamo l’attributo RECEIVING DATE per produrre un archivio gerarchico:

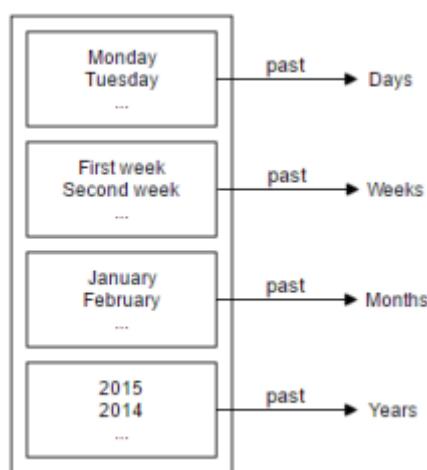


Figura 1.55: Whatsapp: Schema gerarchico costruito sulla base dell’attributo Receiving Date

Considerando il caso di una email software, si possono organizzare le email ricevute per categorie: anni, mesi, settimane, giorni ecc. Si mette quindi più attenzione alle email ricevute di recente rispetto alle più vecchie (viene assegnato un ranking basato sul tempo).

EXAMPLE

consideriamo il seguente scenario

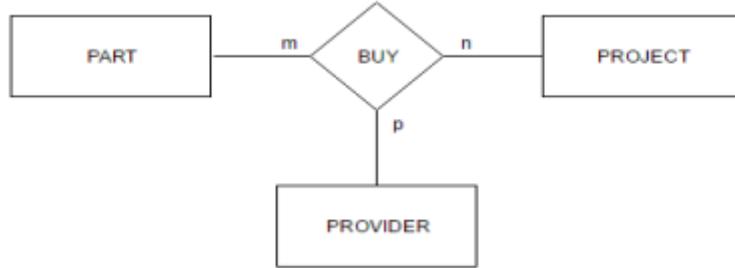


Figura 1.56: Relazione ternaria con PART, PROJECT, PROVIDER

Possiamo decidere di usare una relazione ternaria perché tre relazioni binarie non sono sufficienti per dire che una parte specifica è stata fornita da uno specifico fornitore ad uno specifico team (o project). Una relazione ternaria quindi collega i tre oggetti nello stesso tempo, cosa che non succederebbe se usassimo tre relazioni binarie. Per dimostrare che la quantità di informazioni è variata possiamo usare il concetto di reificazione. Per trasformare una relazione abbiamo infatti bisogno di un nuovo schema in cui compaiono tre relazioni binarie più una nuova entità (PROVISIONING):

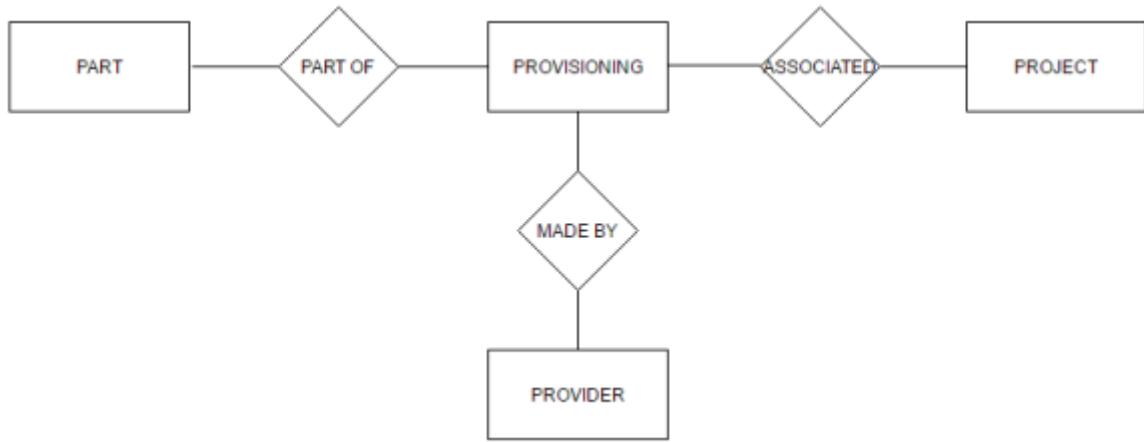


Figura 1.57: Reificazione della Relazione ternaria con PART, PROJECT, PROVIDER

Per verificare la cardinalità delle partecipazioni di ogni entità alla relazione ternaria, focalizziamo il nostro punto di vista su ciascuna di esse, vedendo le parti rimanenti del DB come una coppia. Ad esempio, concentrandoci sull'entità PART la relazione con la coppia (PROJECT,

PROVIDER) è di tipo a molti (n), in quanto ci sono più parti che possono essere fornite da differenti fornitori per differenti progetti. Nello stesso modo, concentrandoci sull'entità PROJECT e osservando la relazione con la coppia (PART, PROVIDER), si nota che in un progetto si possono utilizzare più parti diverse da diversi fornitori, pertanto si assegna una relazione a molti (m). Nel caso del PROVIDER la relazione è di tipo a molti (p), perché si possono vedere una o più coppie del tipo (PART, PROJECT). Bisogna tener presente che una volta effettuata la reificazione la cardinalità non varia.

Supponiamo di avere due entity type con la seguente cardinalità:



Figura 1.58: Customer Buys Product

Trasformando tale schema si ricaveranno tre tabelle, due per le entità e un'altra per la relazione (in quanto si ha una cardinalità n:m). Considerando la seguente variante derivata dalla reificazione:



Figura 1.59: Enhanced Customer Buys Product

si è trasformato la relazione BUY nell'entità SHOPPING SESSION che può essere effettuata da un cliente e che può includere più prodotti. Dal punto di vista delle informazioni potremmo anche chiamare tale entità con il nome di BILL O TICKET, ma non è propriamente corretto in quanto non è un oggetto esistente, ma solo una collezione di informazioni di altri oggetti. Tale entità è il risultato della reificazione e non dall'analisi del problema, per tale motivo questa entità si può considerare come una “entity type di seconda classe” perché non è realmente una classe autonoma dalle altre. Una buona modellazione è considerare tale entity type come una weak entity con un proprio customer. Il concetto generale di weak entity type è un po’ differente da quello adottato in questo caso, perché un’entità del genere non può essere identificata in modo univoco dai suoi soli attributi, ma dalla primary key dell’entità collegata ad essa, cioè quella del customer (esempio: due notebook identici sono perfettamente uguali nei loro attributi, ciò che li distingue è il proprietario a cui appartengono). In questo caso la SHOPPING SESSION differisce in parte dal concetto di entity type dal momento che possiede anche attributi riguardanti il ticket number e la data del ticket, pertanto ha già delle sue informazioni caratteristiche (la sua primary key che la identifica). Infine un CUSTOMER può avere più SHOPPING SESSION, ma ciascuna di esse può avere solo un CUSTOMER e pertanto la relazione a molti (n) dello schema di partenza diventa 1:n. D’altra parte una SHOPPING SESSION può includere più

prodotti come anche più prodotti possono essere inclusi in più SHOPPING SESSION facendo diventare la relazione a molti (m) una relazione n:m. Quindi la cardinalità dello schema ER di partenza è legata a quella specificata dopo la reificazione. Gli attributi della relazione di partenza diventeranno gli attributi dell'entità derivata dalla reificazione.

EXERCISE: PHR (EHR)

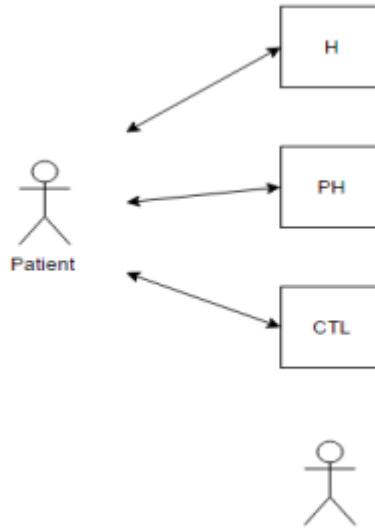


Figura 1.60: Personal Health Record (PHR)

(proviamo ad usare PRICE CHAIN e ARCHIVE PATTERN) Questo problema è chiamato PHR (Personal Health Record) (Cartella Sanitaria Elettronica): il paziente si reca dal medico di famiglia, viene informato circa la sua situazione clinica e gli vengono prescritti alcuni test da fare. Il paziente si reca al centro dei test clinici (CTL), oppure in ospedale (H, PH) per radiografie o per altri test. Successivamente i risultati verranno consegnati al medico di famiglia. Questo è uno scenario classico. Perché non creare un database online di pazienti, così da mettere tutti i loro report e accedere alle informazioni usando solo username e password? Il problema è abbastanza semplice da descrivere, ma è abbastanza complesso da risolvere. Google creò alcuni anni fa Google Health e Microsoft presentò Microsoft HealthVault, due applicazioni che permettevano di inserire tutte le nostre informazioni nel database. Google e Microsoft hanno realizzato questi database e hanno fallito. Adesso non ci sono soluzioni disponibili in questo ambito.

Possibili problemi in cui si può incorrere sono:

- persone anziane;
- no standard;
- benefits misunderstanding (i pazienti non comprendono i benefici);
- i dottori non vogliono l'automazione (power knowledge);
- aspetti legati alla privacy.

Il fallimento di tali sistemi sta anche nel fatto che potremmo essere “bombardati” da tanti annunci riguardanti alcuni specifici rimedi. Se qualcuno pagasse, potrebbe forzare Google a dare informazioni false riguardo la nostra salute. Inoltre le preoccupazioni relative alla privacy sono estremamente importanti e l’interesse economico del mondo dei dati personali della salute è di decine di miliardi di euro.

Analizziamo i requisiti del database di una clinica.

Quando il paziente arriva in clinica, la receptionist lo ammette e informa il dottore che è arrivato un paziente (è una ammissione tecnica, gli permette l’ingresso). Il dottore a sua volta ammette il paziente che sta aspettando, questa è un’ammissione clinica, perché dal punto di vista medico un dottore dice al paziente che può iniziare una cura/terapia. Il paziente quindi descrive il problema e il dottore fa l’anamnesi (scrive le note e mette tutte le informazioni nella cartella del paziente, nel sistema). I dottori sottoscrivono la prescrizione, i test e invitano i pazienti a sottoporsi ad essi (anche nella stessa clinica). Successivamente delle infermiere effettuano i test e i risultati sono poi inseriti nel sistema. I pazienti potranno visualizzare i risultati dei test a casa sul loro computer. Elenchiamo i tipi di utente presenti nello scenario descritto:

- DOCTOR;
- RECEPTIONISTS;
- PATIENTS;
- NURSES.

Descriviamo lo scenario mediante questo semplice schema:

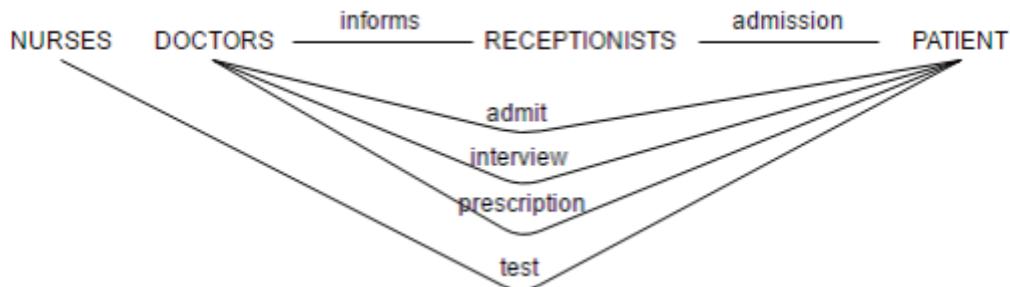


Figura 1.61: Clinics - Requirements

Possiamo vedere quanto facile e immediato è descrivere un problema in termini di dati scambiati tra i tipi di utente. Il framework che abbiamo visto è molto efficace perché iniziamo a vedere ciò che ci circonda in modo differente. Ci sono: tipi di utente, azioni, informazioni (che sono scambiate tra i vari utenti), tipi di relazioni tra entità. Abbiamo così una buona e immediata descrizione di un problema reale.

Iniziamo una prima fase di modellazione considerando tre relazioni binarie tra le seguenti entità:

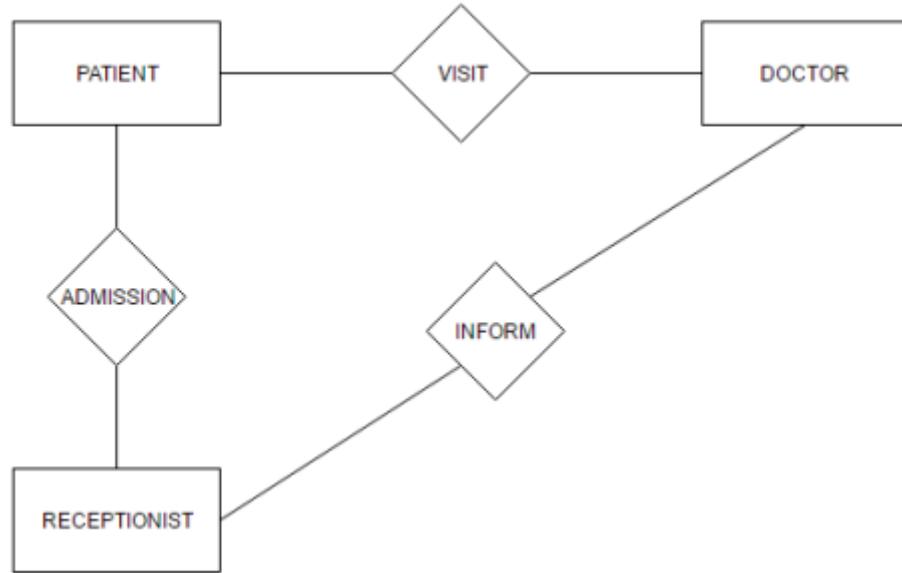


Figura 1.62: Clinics - ER Draft

Andrea Cuna
Giuseppe Levantaci
13/10/2016

Riprendiamo la modellazione del problema della clinica. Sono sicuramente necessarie le seguenti entity type:

- PATIENT;
- DOCTOR;
- RECEPTIONIST.

Dobbiamo tener conto del fatto che un paziente prenota in anticipo una visita. Quando egli si presenta in ambulatorio, dovrà essere “ammesso” alla visita dal receptionist, il quale ovviamente “informerà” il medico:

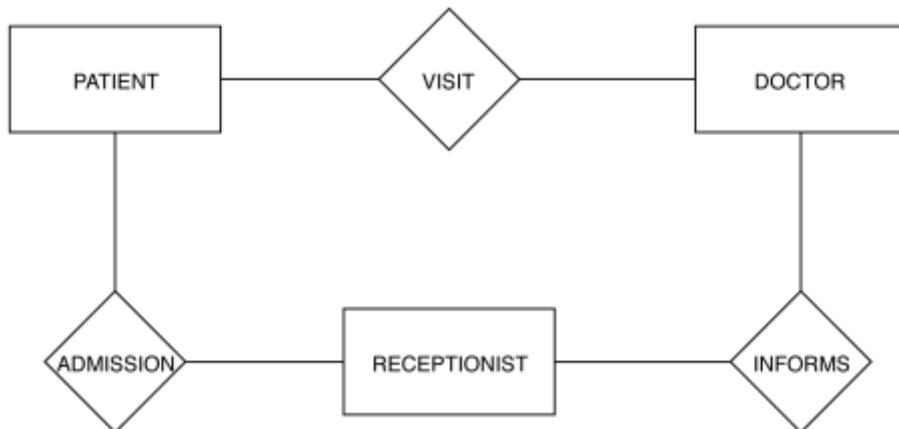


Figura 1.63: Clinics - Incremental ER

Vediamo ora come specificare l'intervista del dottore al paziente:

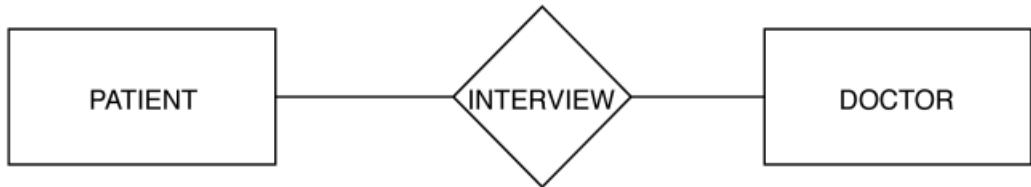


Figura 1.64: Doctor Interviews Patient

Va ricordato, ora, che le interviste riguardano la storia clinica del paziente, lo stile di vita e il problema medico attuale, e l'assunzione o meno di farmaci, per cui avremo le seguenti relazioni:

- Il paziente è connesso ad altri pazienti da una relazione (ricorsiva) di parentela (informazione utile per considerare l'eventuale ereditarietà delle malattie):

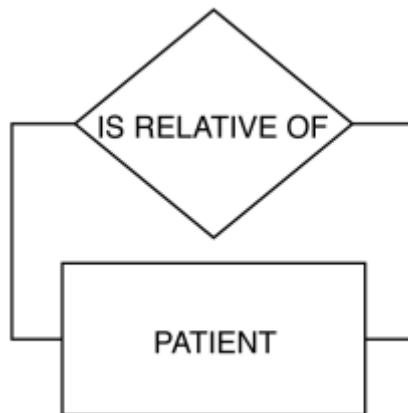


Figura 1.65: Patient Is Relative Of Patient

- Il paziente è malato:

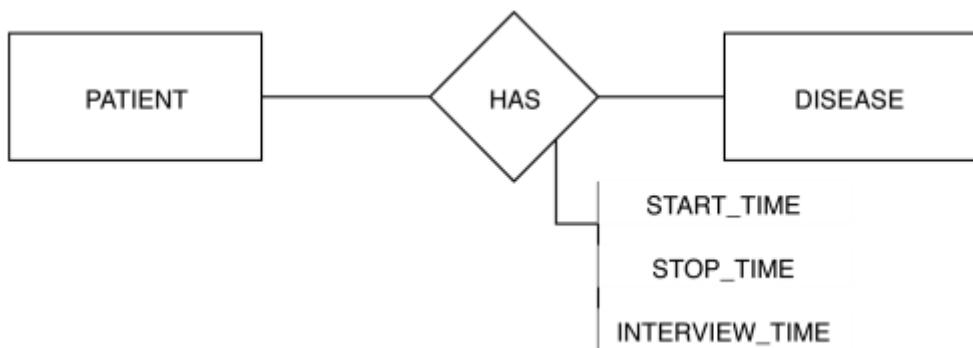


Figura 1.66: Patient Has Disease

- Il paziente assume medicinali:

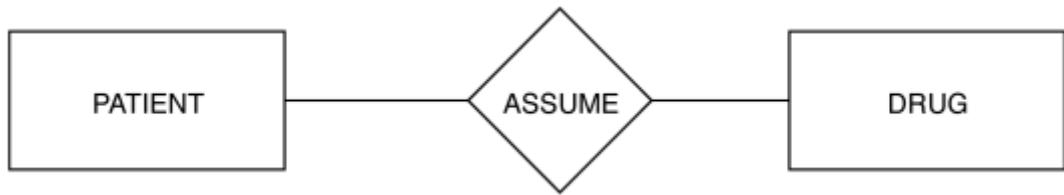


Figura 1.67: Patient Assume Drug

- Ora inseriamo il concetto di stile di vita del paziente (ad esempio se è un fumatore) modellando una nuova entity type:

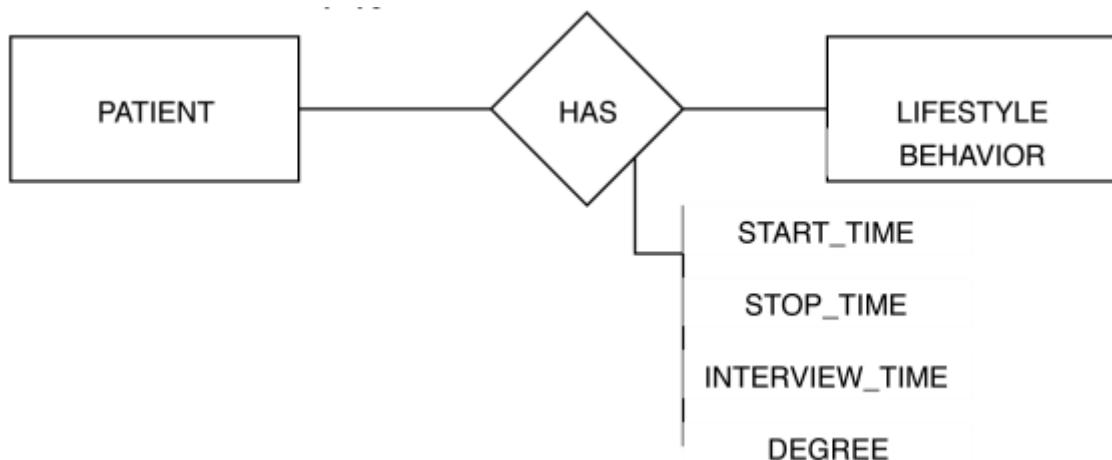


Figura 1.68: Patient Has Lifestyle Behavior

Come si può vedere, queste ultime quattro relazioni tengono traccia delle informazioni che scaturiscono dall'intervista.

Vediamo ora come si può modellare la prescrizione di medicine, esami, test e indagini cliniche al paziente da parte del dottore:

- Si potrebbe modellare in questo modo:



Figura 1.69: Patient Has Lifestyle Behavior

- Potremmo considerare anche:

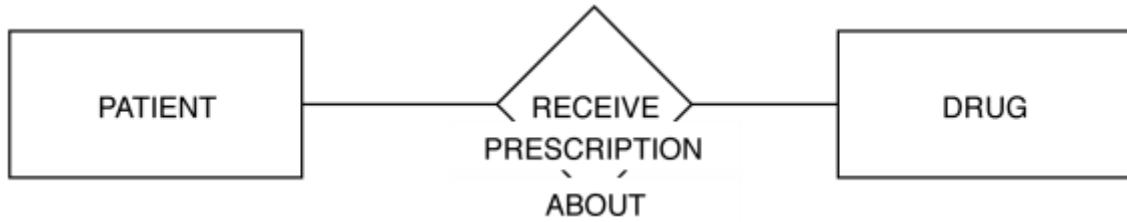


Figura 1.70: Patient Receives Prescription About Drug

Ora sappiamo che durante la visita il dottore prescrive le medicine al paziente, inoltre l'intervista è qualcosa che avviene durante la visita. Per questo è possibile applicare la reificazione a VISIT e assumere che durante la visita il paziente dichiari di usare medicine e di avere un certo stile di vita; in questo modo non sarà più necessario specificare la data dell'intervista come attributi delle due relazioni HAS, e si terrà anche conto del dottore.

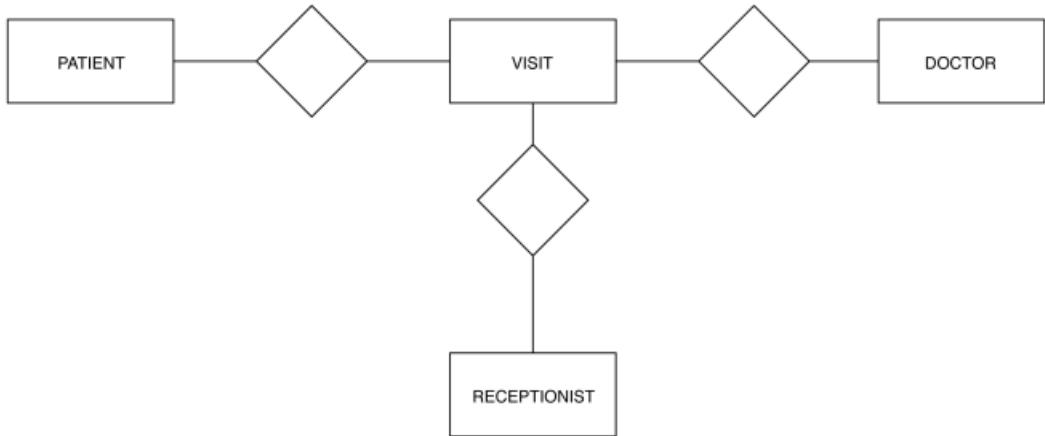


Figura 1.71: Clinics - Incremental ER 2

In precedenza abbiamo detto che durante l'intervista (e quindi durante la visita), il paziente comunica al dottore le medicine che assume, il dottore prescrive al paziente delle medicine e gli diagnostica una malattia. Queste considerazioni ci permettono di associare DISEASE e DRUG a VISIT, invece che a PATIENT.

In questo modo le relazioni precedenti diventano:

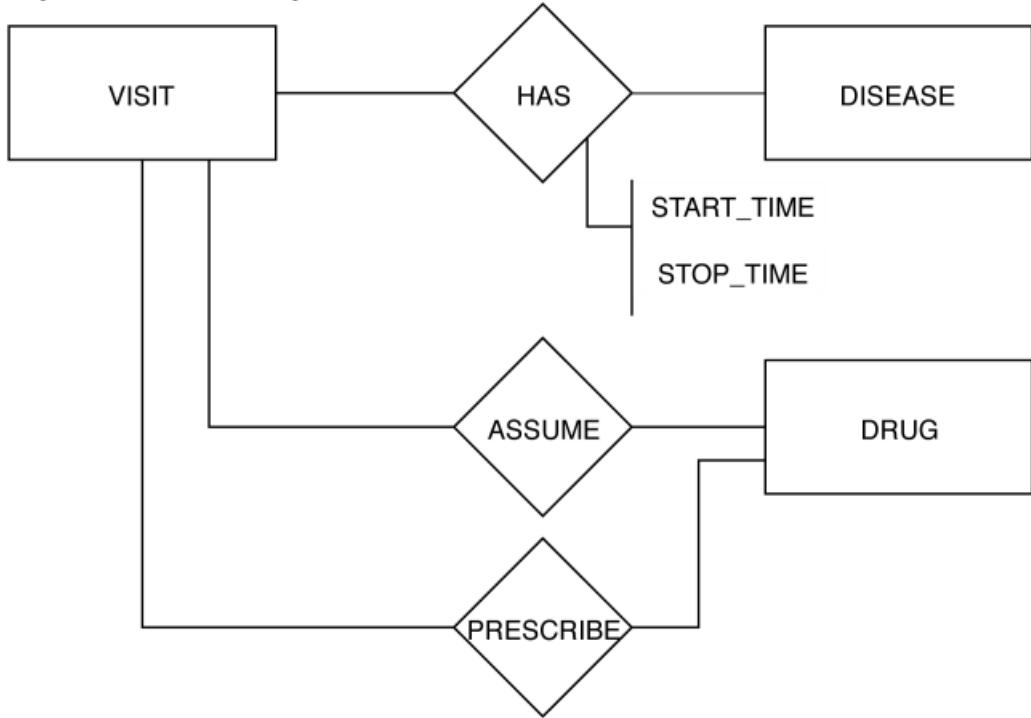


Figura 1.72: Clinics - Incremental ER 3

In seguito ad una visita, il medico potrebbe anche prescrivere al paziente un “esame”, il quale verrà effettuato da un “infermiere”, quindi possiamo modellare in questo modo:

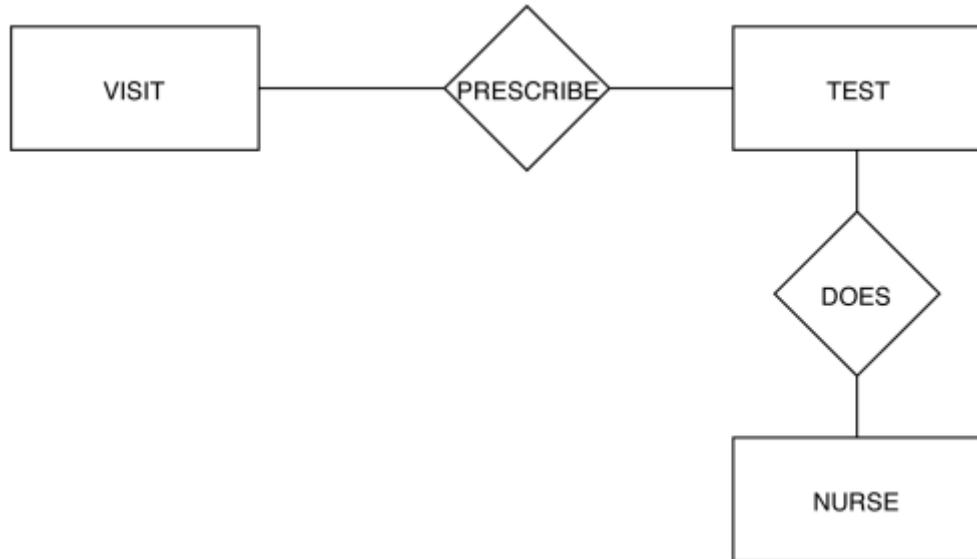


Figura 1.73: Clinics - Incremental ER 4

In particolare, **NURSE** effettua un **TEST** (nel senso di un esame specifico di un determinato paziente) di tipo **TYPE OF TEST** (inteso come esame in senso generico, ad esempio “analisi

del sangue”), il quale è prescritto da un medico (stiamo utilizzando il pattern Object-Type of object). Abbiamo bisogno dunque delle seguenti relation types:

- DOES (NURSE-DOES-TEST);
- INSTANCE OF (TEST-INSTANCE OF-TYPE OF TEST);
- PRESCRIBES (VISIT-PRESCRIBES-TYPE OF TEST).

La prescrizione del test si può modellare, dunque, in questo modo:

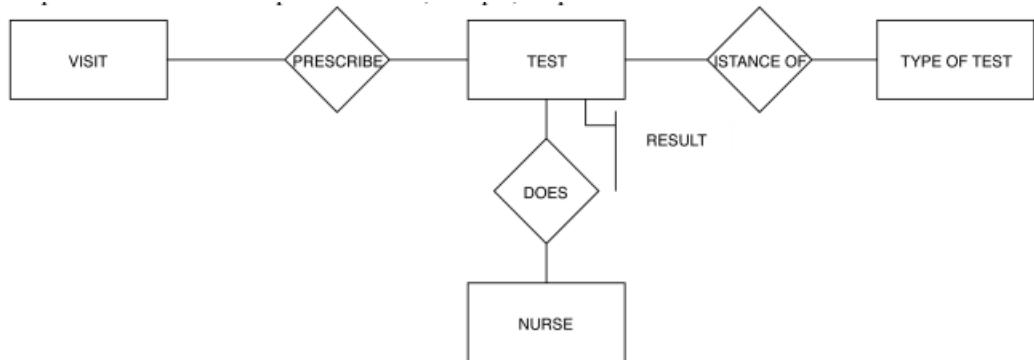


Figura 1.74: Clinics - Incremental ER 5

Il modello ER finale per il nostro problema è il seguente:

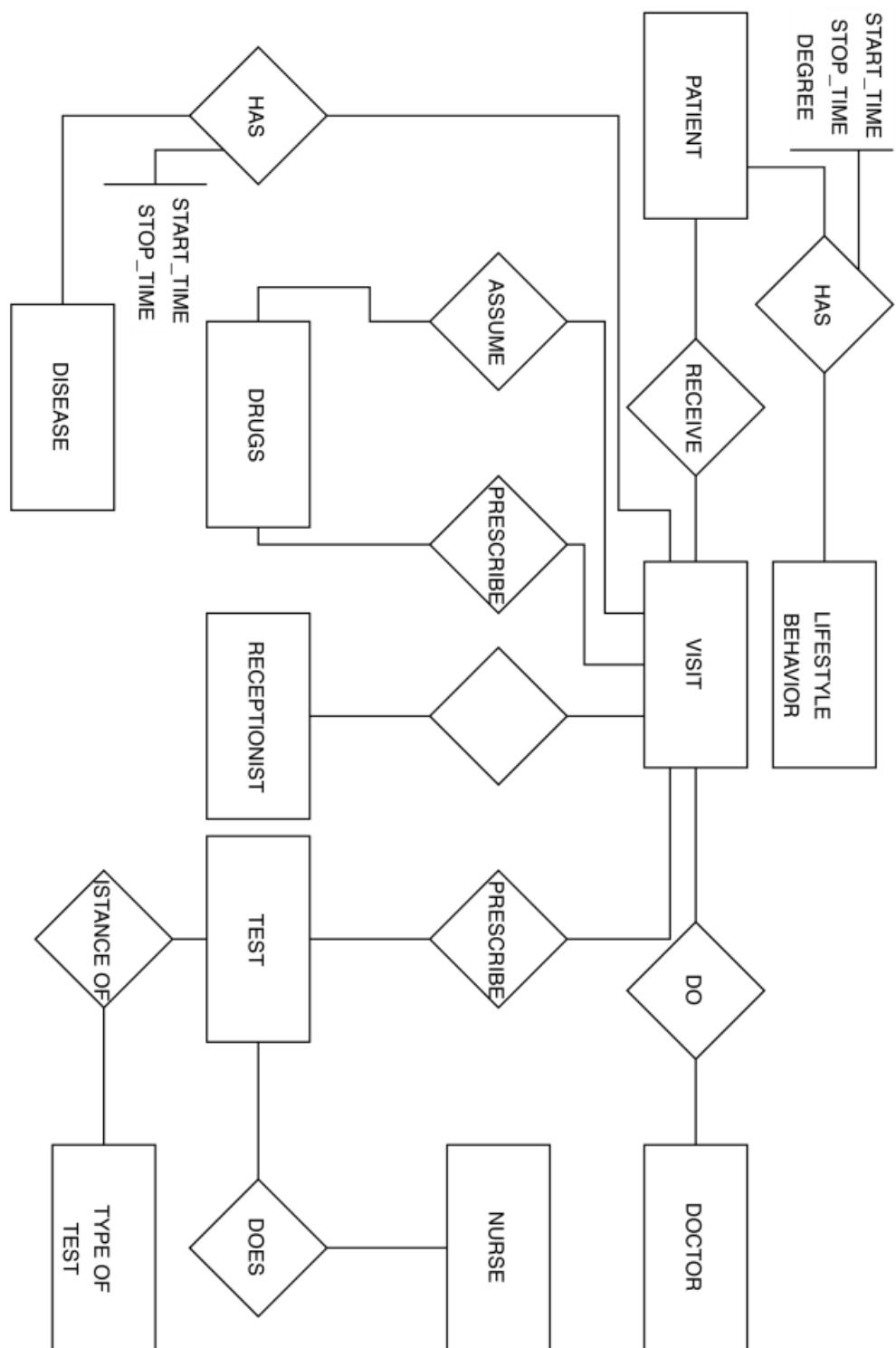


Figura 1.75: Clinics - Final ER

1.9 DBMS

1.9.1 Descrizione dei principali DBMS

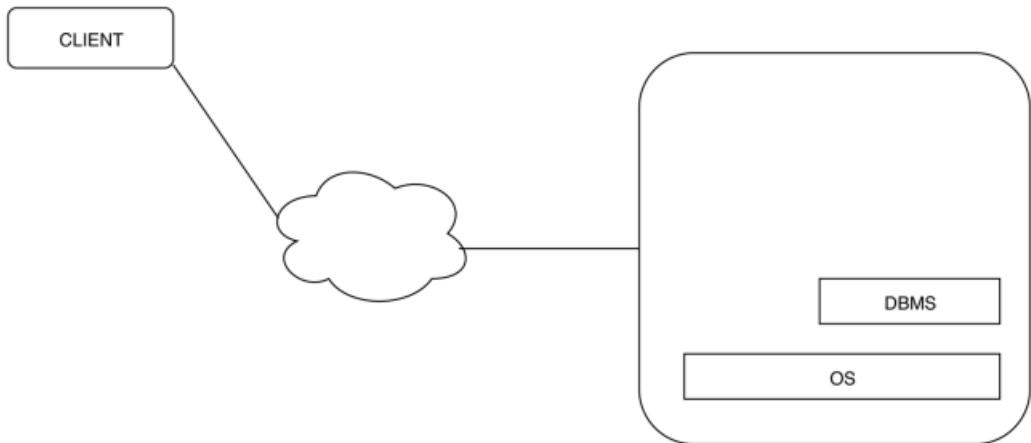


Figura 1.76: DBMS

Tra i DBMS più conosciuti troviamo:

- MySQL;
- MongoDB;
- SQLite;
- PostgreSQL;
- Oracle;
- Microsoft SQL Server;
- Microsoft Access (e il suo competitor OpenOffice Base);
- Couchbase.

Analizziamone alcuni più in dettaglio.

- **Oracle, Microsoft SQL Server e MySQL** sono RDBMS, ovvero DBMS “relazionali” (che coprono circa il 90 % del mercato dei DBMS). Oracle ha un prezzo di circa 60.000 €, SQL Server 10.000 €, mentre MySQL è gratuito, pur essendo in qualche modo paragonabile agli altri due. Oracle e SQL Server sono di fascia “enterprise”, impiegati quindi nell’ambito di grandi aziende (es. banche), mentre MySQL è perfetto per gli studenti universitari che si affacciano al mondo dei database;
- **MongoDB** e **Couchbase** sono DBMS “NoSQL”, ovvero NON relazionali, idonei per particolari applicazioni;
- **PostgreSQL** è un Object Relational DBMS, ovvero presenta alcune caratteristiche tipiche della programmazione orientata agli oggetti. Questo consente un Object-Relational Mapping più preciso;

- Infine, **Microsoft Access** e il suo diretto competitor open source **OpenOffice Base** sono semplici DBMS utilizzati principalmente per la costruzione veloce di prototipi di database.

1.9.2 Installazione e configurazione di MySQL

- *Installazione su Microsoft Windows*

Recarsi al seguente indirizzo: <http://dev.mysql.com/downloads/mysql/>; Selezionare il proprio Sistema Operativo ed eseguire il Download.

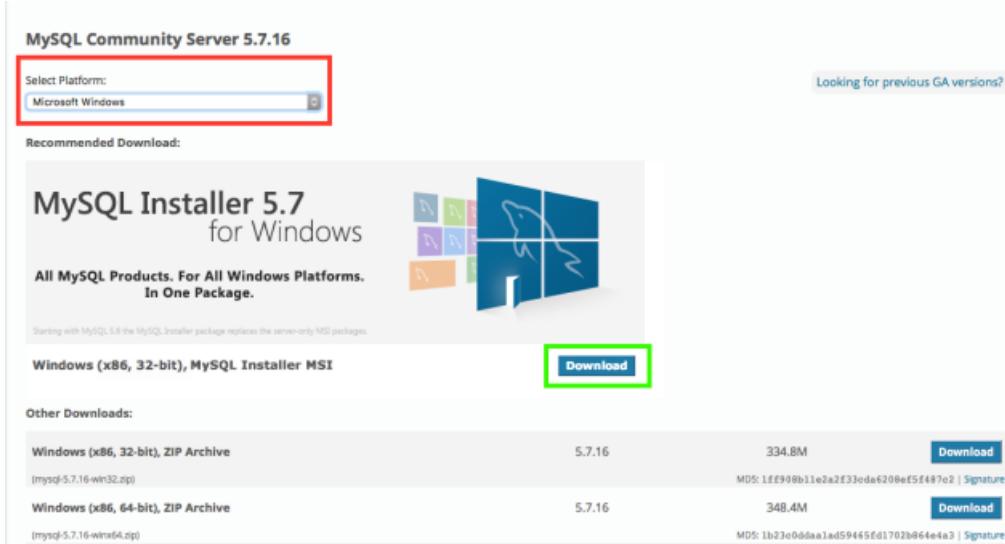


Figura 1.77: MySQL Installer Download

Prima di effettuare l'installazione eseguire il seguente download:

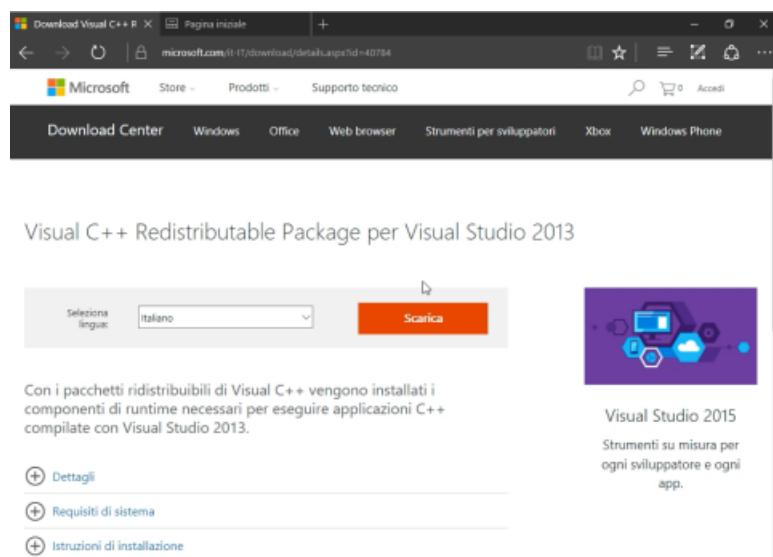


Figura 1.78: Visual C++ Redistributable Packages download

Dopo il download selezionare ed avviare l'installazione, il file avrà nome: "mysql-installercommunity-x.x.x.x-dmr.msi" dove x.x.x.x sarà il numero di versione, verrà poi visualizzata la seguente schermata:



Figura 1.79: MySQL Installer Splash-Screen

Verrà visualizzata la seguente schermata:

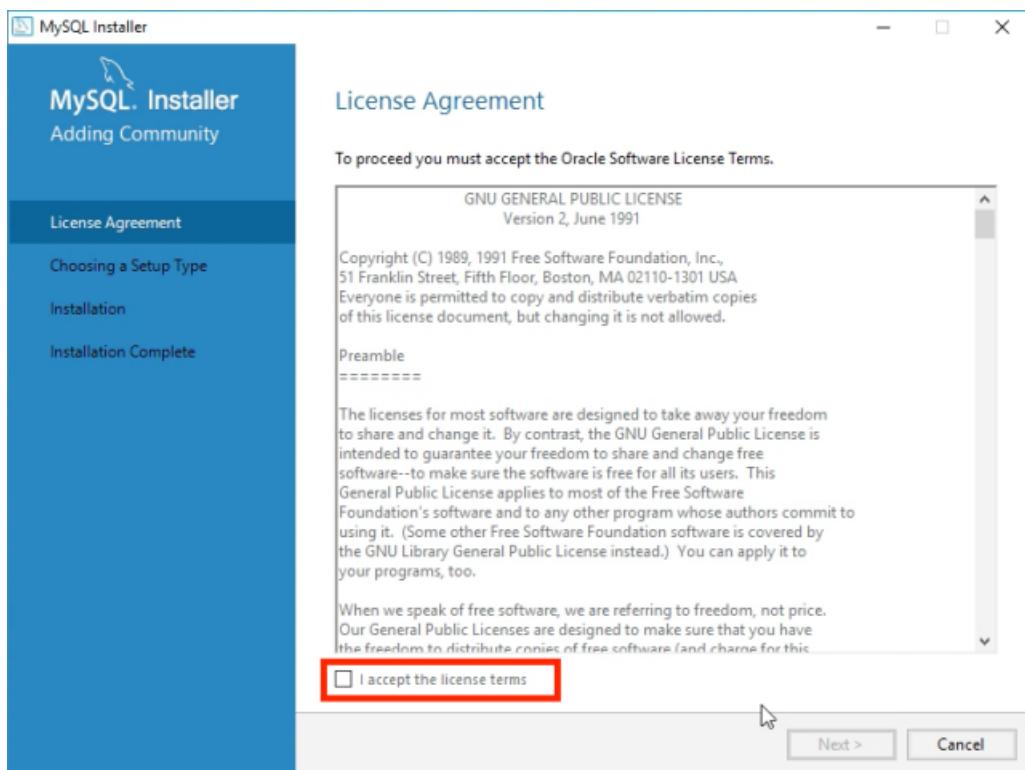


Figura 1.80: MySQL Installer Wizard

Accettare i termini di licenza e poi proseguire nell'installazione.

Nella schermata successiva selezionare quanto evidenziato:

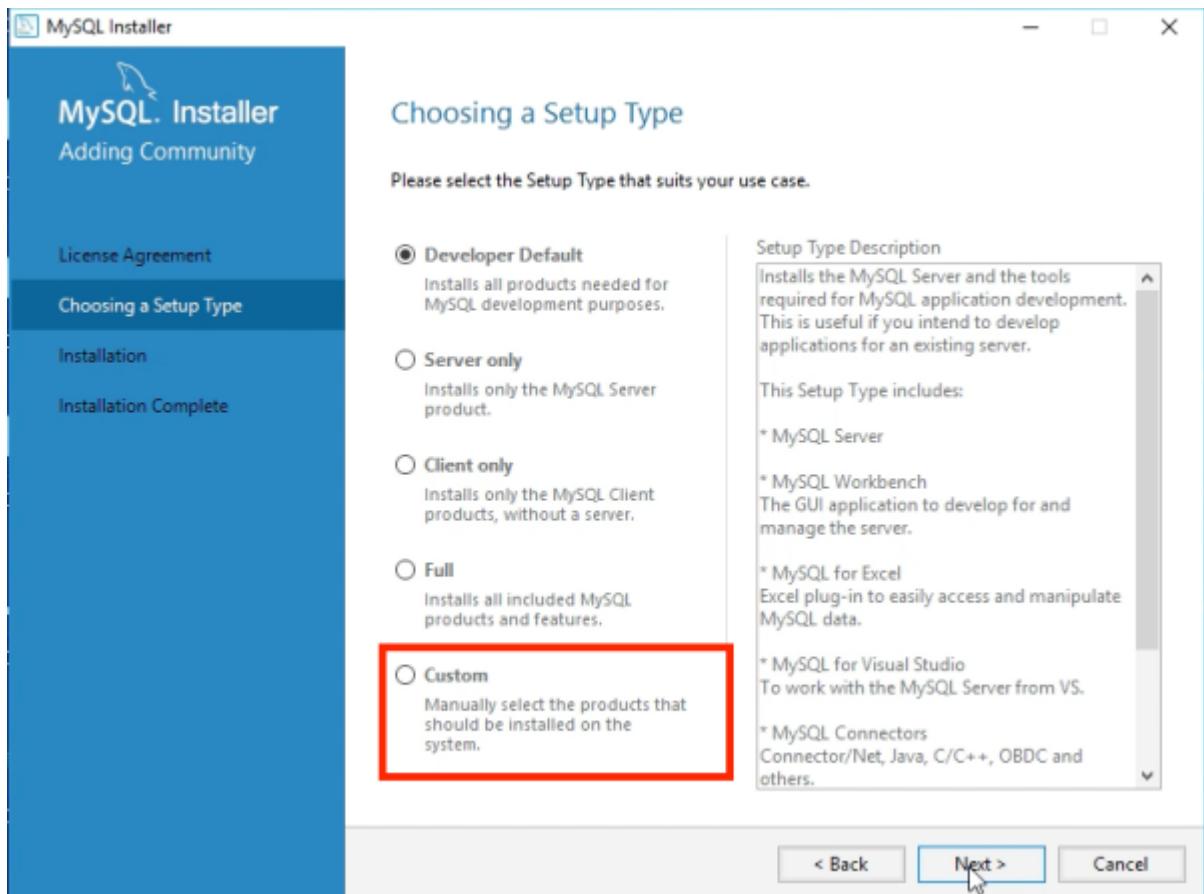


Figura 1.81: MySQL Installer Wizard 2

E nella schermata successiva configurare i pacchetti come riportati in figura:

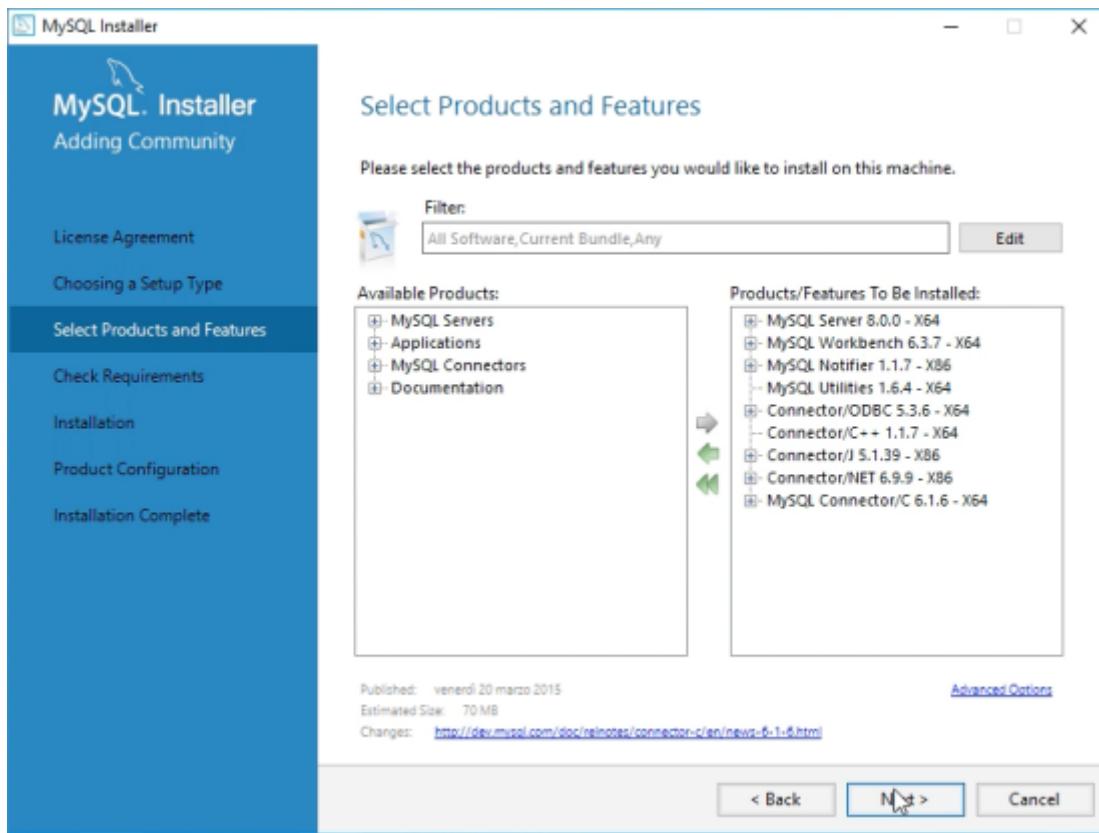


Figura 1.82: MySQL Installer Wizard 3

e quindi proseguire nell'installazione cliccando due volte sul tasto “Next >”. Si raggiungerà la seguente schermata:

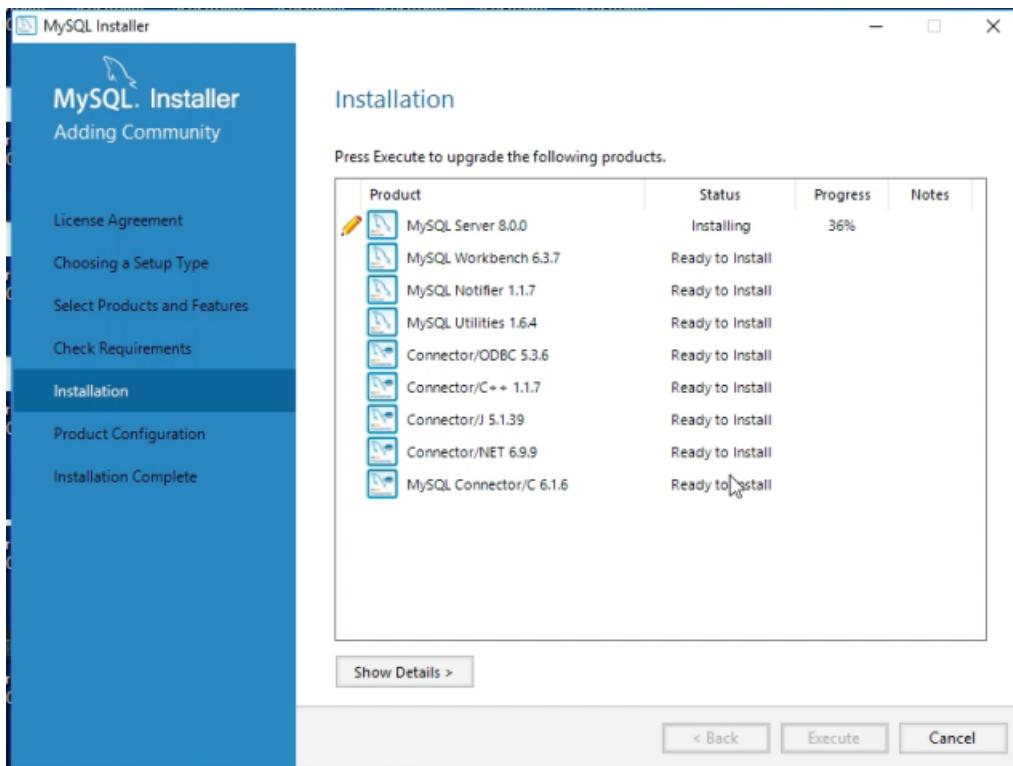


Figura 1.83: MySQL Installer Wizard 4

Al termine dell’installazione verrà installato, oltre al server, il software “MySQL Work-Bench” il quale è utile per creare e gestire le tabelle dei database contenuti nel server. Al primo avvio del server verrà richiesto di inserire una password di amministratore: qui si può inserire una password a scelta che dovrà essere ricordata per eseguire l’accesso negli usi successivi.

- *INSTALLAZIONE SU MACOS*

Recarsi allo stesso URL indicato in precedenza, selezionare il sistema operativo MacOS ed eseguire il download del pack dmg. In seguito, una volta aperto il pack dmg ed avviata l’installazione del pack pkg al suo interno, si otterrà questa schermata:

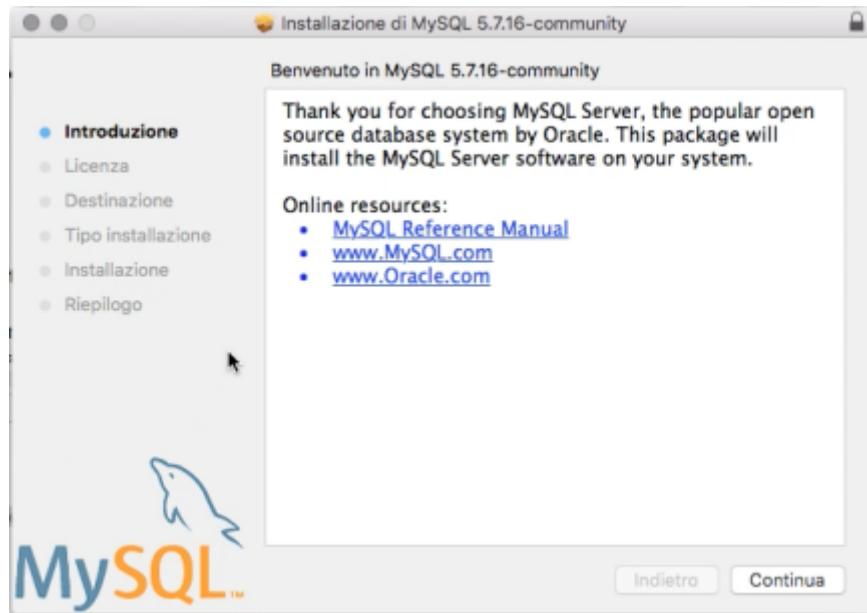


Figura 1.84: MySQL Installer MacOS Wizard

cliccando su continua si dovrà poi accettare la licenza:



Figura 1.85: MySQL Installer MacOS Wizard 2

Successivamente nella schermata successiva basterà cliccare su “Installa” e verrà avviata l’installazione durante la quale sarà richiesta la password di amministratore:

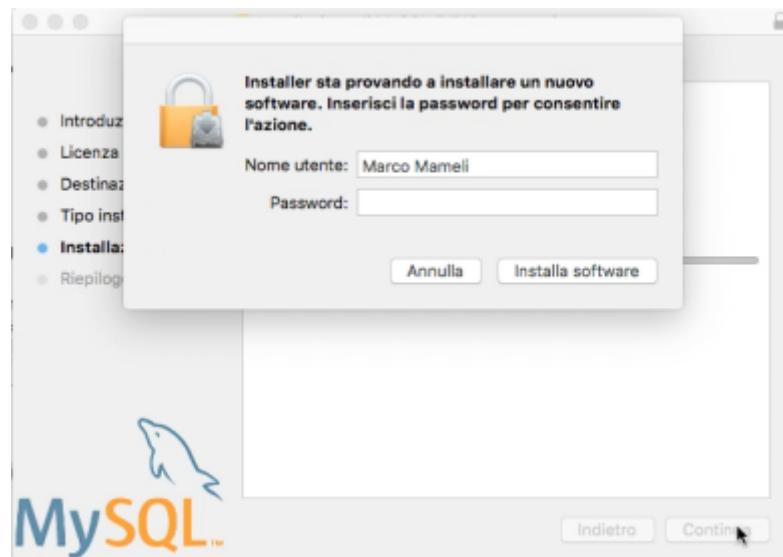


Figura 1.86: MySQL Installer MacOS Wizard 3

Alla fine dell'installazione verrà visualizzata una finestrella contenente la password di amministratore generata automaticamente:

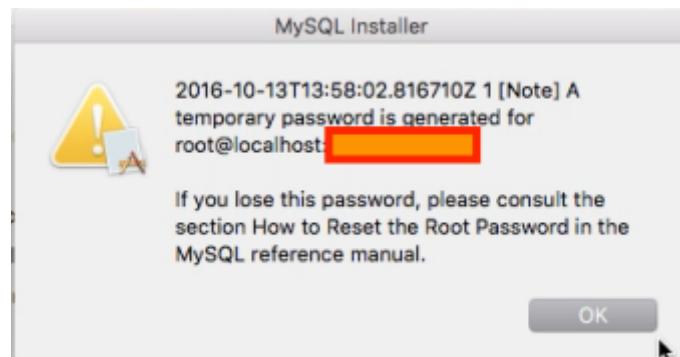


Figura 1.87: MySQL Installer MacOS Wizard 4

Dopo di che sarà possibile avviare il Server tramite le impostazioni:

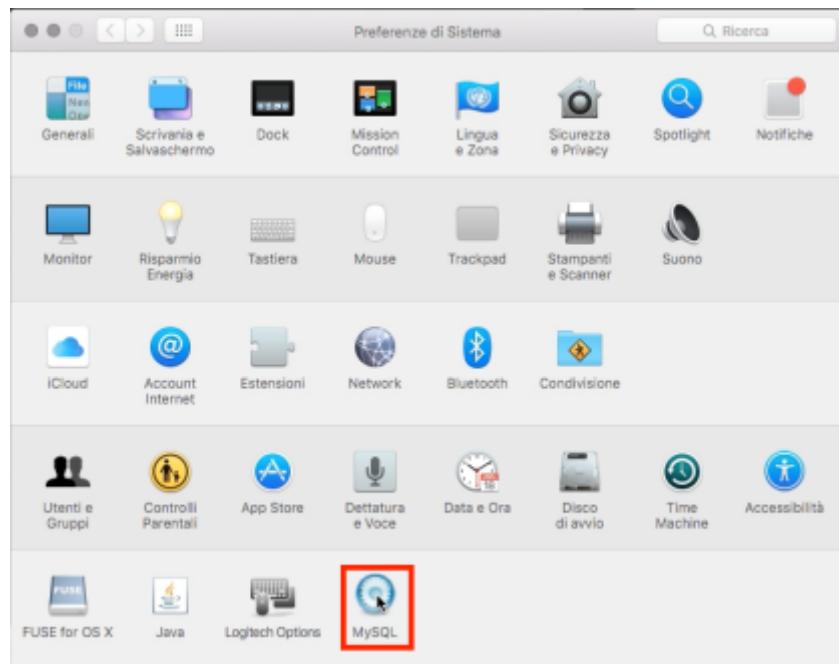


Figura 1.88: Mac OS Settings MySQL

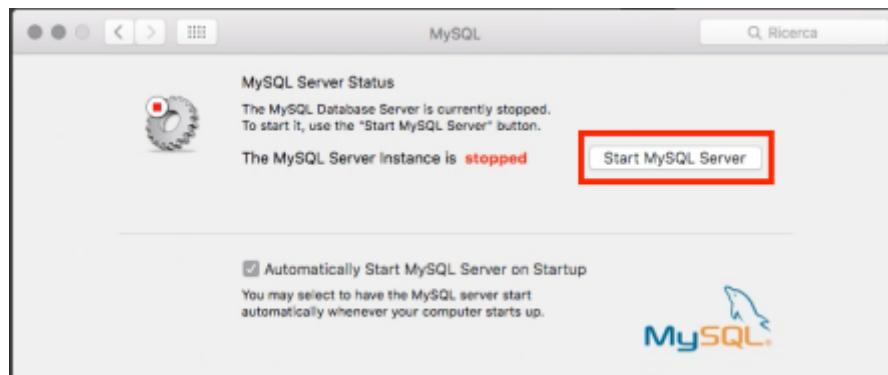


Figura 1.89: MySQL Server Status

Ora bisogna installare “MySQL WorkBench”: Eseguire il download da questo link: <http://dev.mysql.com/downloads/workbench/>. Seguendo i passaggi riportati sopra per la selezione del sistema operativo. Una volta eseguito il download aprire il pack dmg ottenendo:

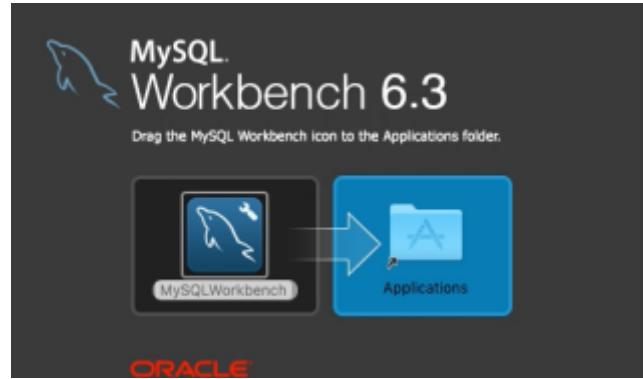


Figura 1.90: MySQL Workbench Splash-Screen

E come suggerisce l’immagine, spostare l’applicazione nella cartella per eseguire l’installazione.

1.9.3 PRIMO CONTATTO CON MYSQL WORKBENCH

Aprire il programma. Si troverà la seguente schermata:

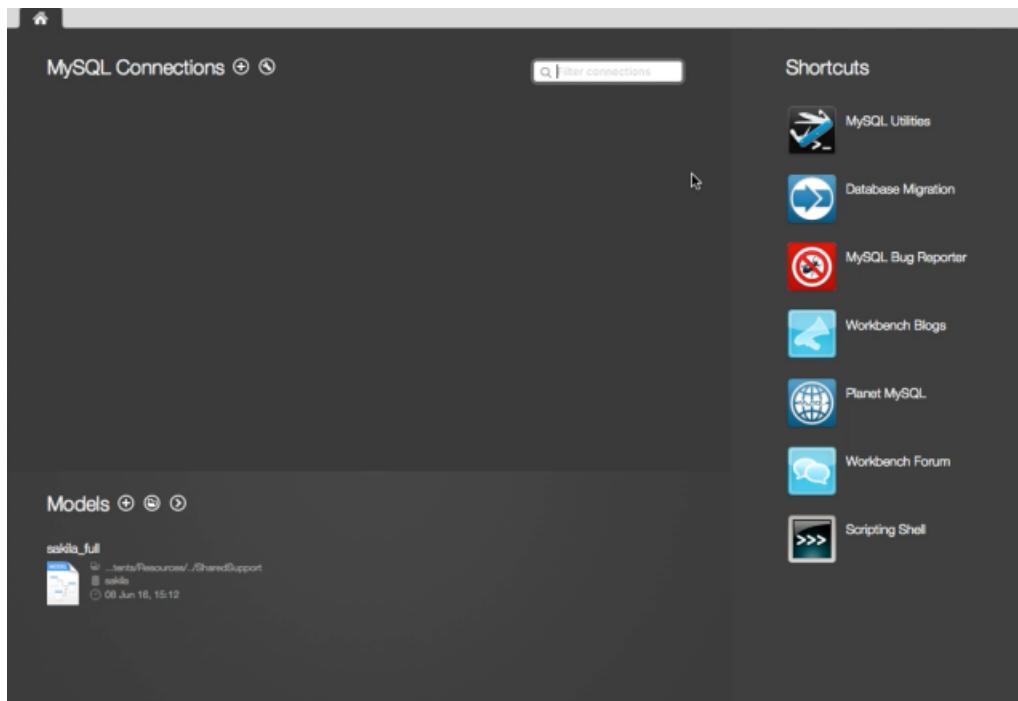


Figura 1.91: MySQL Workbench Home Screen

Cliccando sul “+” accanto a MySQL Connections si potrà aggiungere una nuova connessione ad un database SQL già avviato (vedi passaggi precedenti) visualizzando una schermata che chiederà di inserire la password da amministratore:



Figura 1.92: MySQL Login Screen

Una volta eseguito l'accesso, si avrà la seguente schermata:

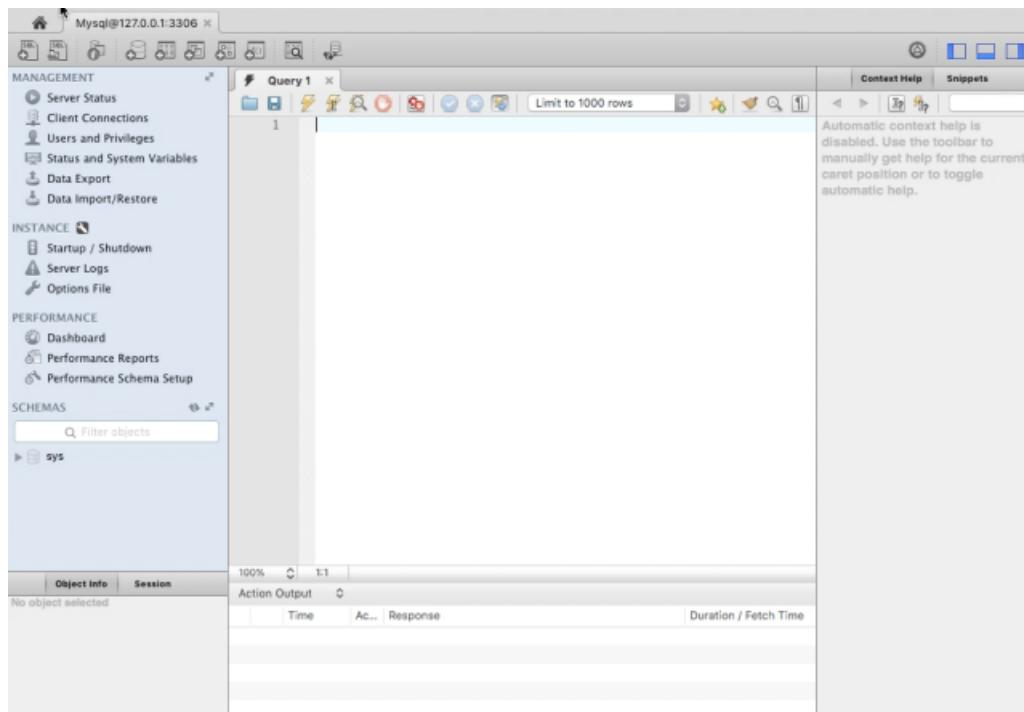


Figura 1.93: MySQL Workbench Initial Screen

Tramite il tasto “Server Status“ è possibile visionare lo stato del server e “Avviare/Fermare” l’istanza del server SQL:

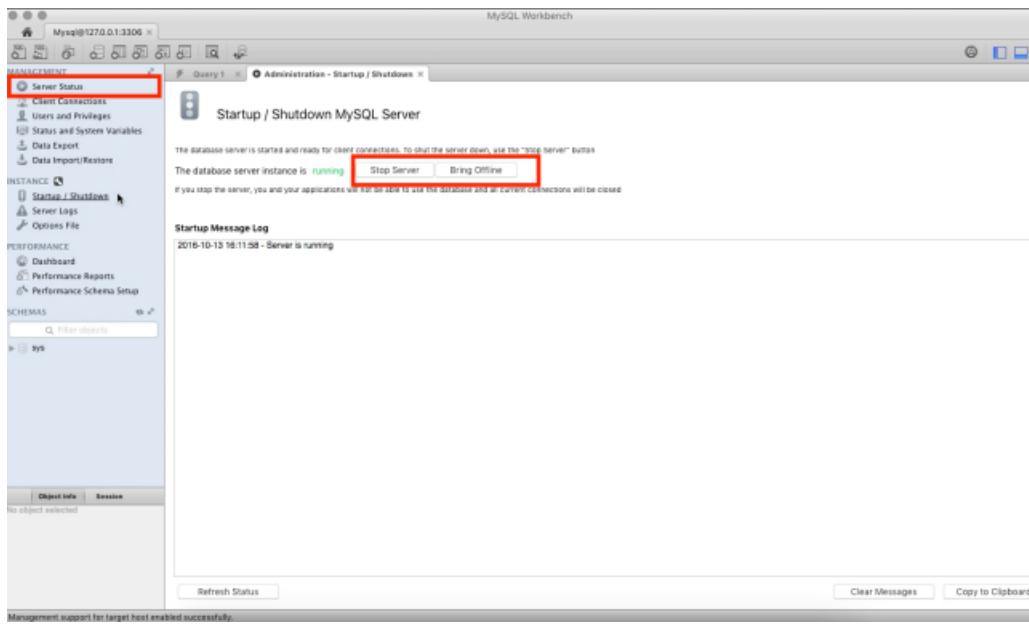


Figura 1.94: MySQL Workbench Server Status

Tramite il tasto “+” posizionato a destra della scritta Models in basso a sinistra: si potrà creare un nuovo modello, visualizzando la seguente schermata:

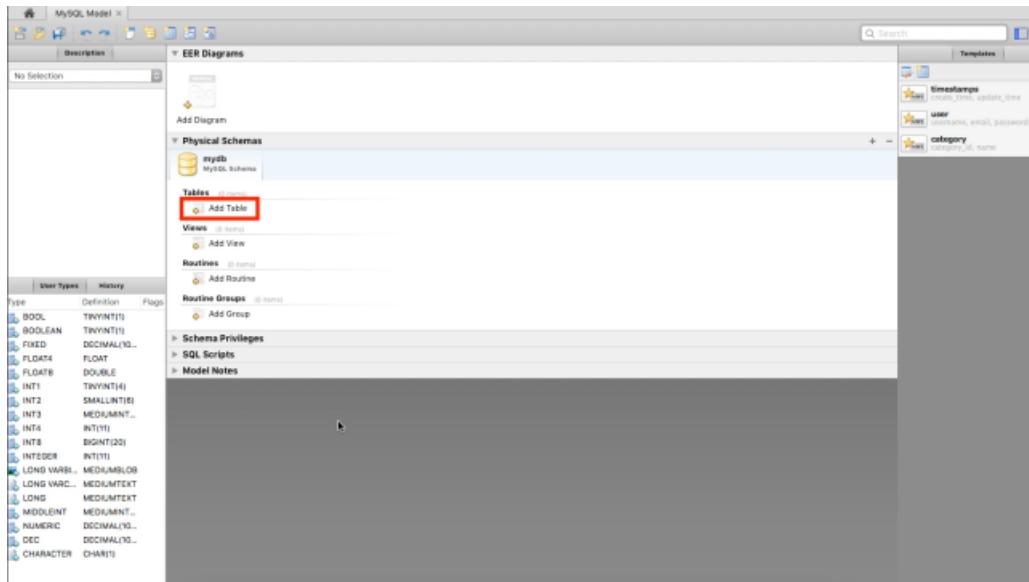


Figura 1.95: MySQL Workbench - Model

Cliccando su “Add Table” (evidenziato nella figura precedente) si potrà inserire una nuova tabella nel nostro modello e la schermata diventerà:

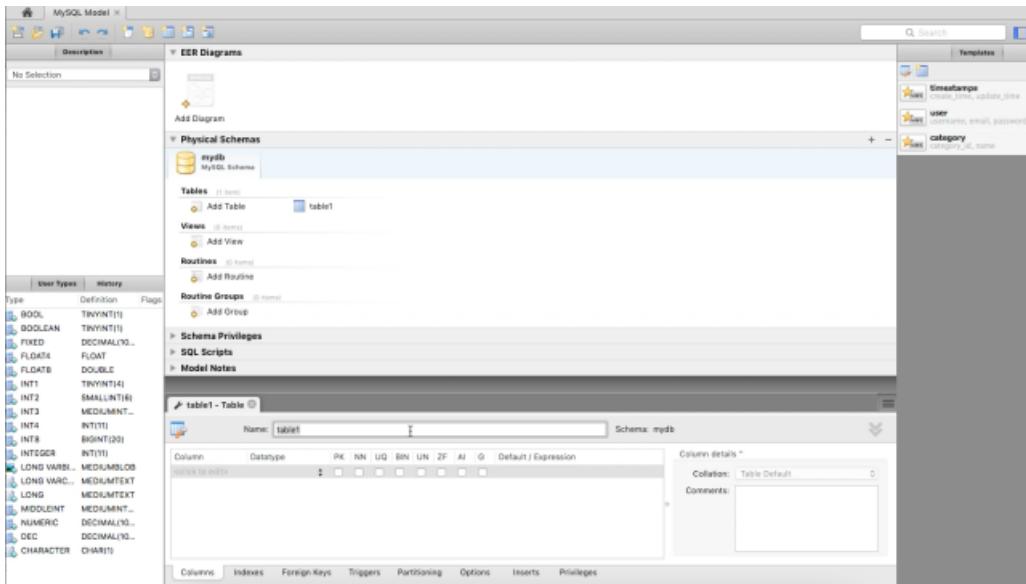


Figura 1.96: MySQL Workbench - Add Table

In basso al centro verrà visualizzata la struttura della nuova tabella. Tramite “Name” sarà possibile rinominarla. Costruire una tabella come quella riportata in figura:

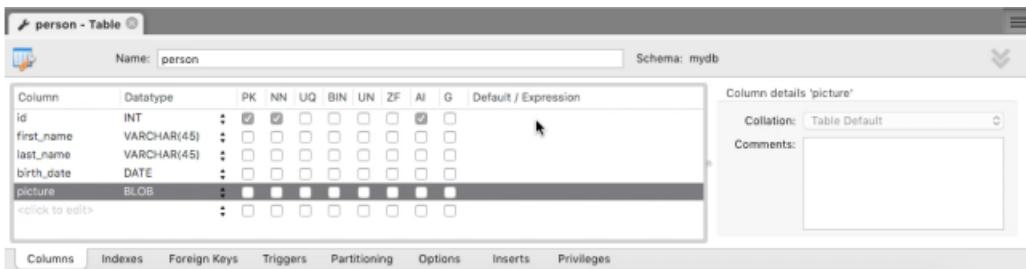


Figura 1.97: MySQL Workbench - New Table

in cui:

- **id** è la CHIAVE PRIMARIA (PK) NON NULLA (NN) e AUTO INCREMENTANTE (AI) identificata come un intero;
- **first_name** e **last_name** sono Nome e Cognome della persona identificati come un insieme di stringhe di massimo 45 caratteri;
- **birth_date** è la data di nascita della persona ed è identificato come un oggetto DATE (ossia una data in formato internazionale AAAA-MM-GG);
- **picture** è invece l’immagine di una persona identificata nel database con un oggetto BLOB ossia un Binary Long Object;

Ora in alto nei menù si dovrà selezionare : “ Database → Forward Engineering ” per eseguire il salvataggio del nostro modello sul server a cui siamo collegati, tramite questo semplice wizard:

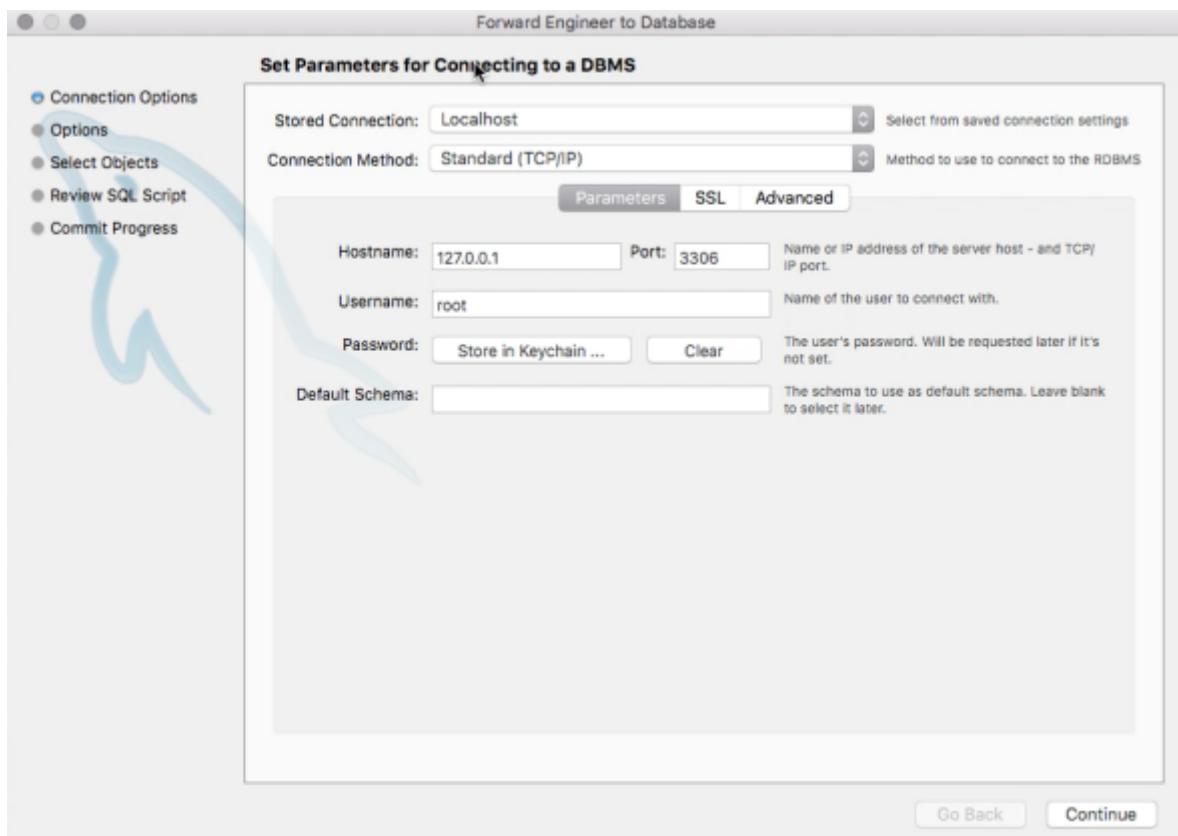


Figura 1.98: MySQL Workbench - Forward Engineering

qui va selezionato il server di destinazione, in seguito selezionare le opzioni necessarie (per ora lasciamo tutto invariato)

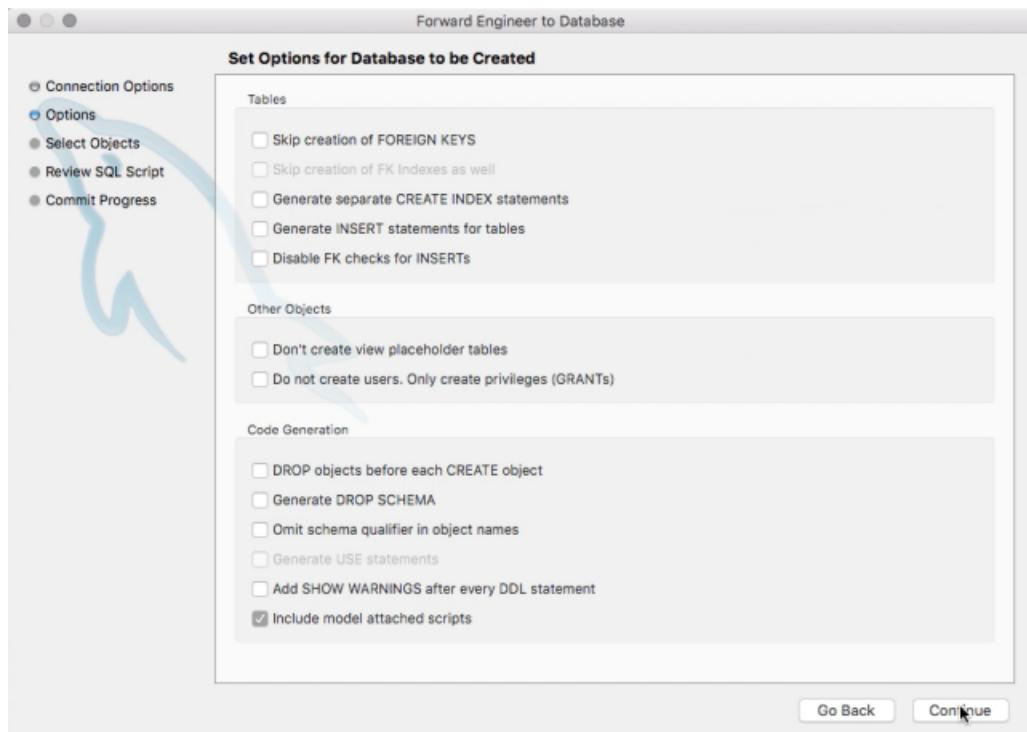


Figura 1.99: MySQL Workbench - Forward Engineering step 2

Poi bisognerà selezionare gli oggetti che si vogliono ottenere da questa esportazione:

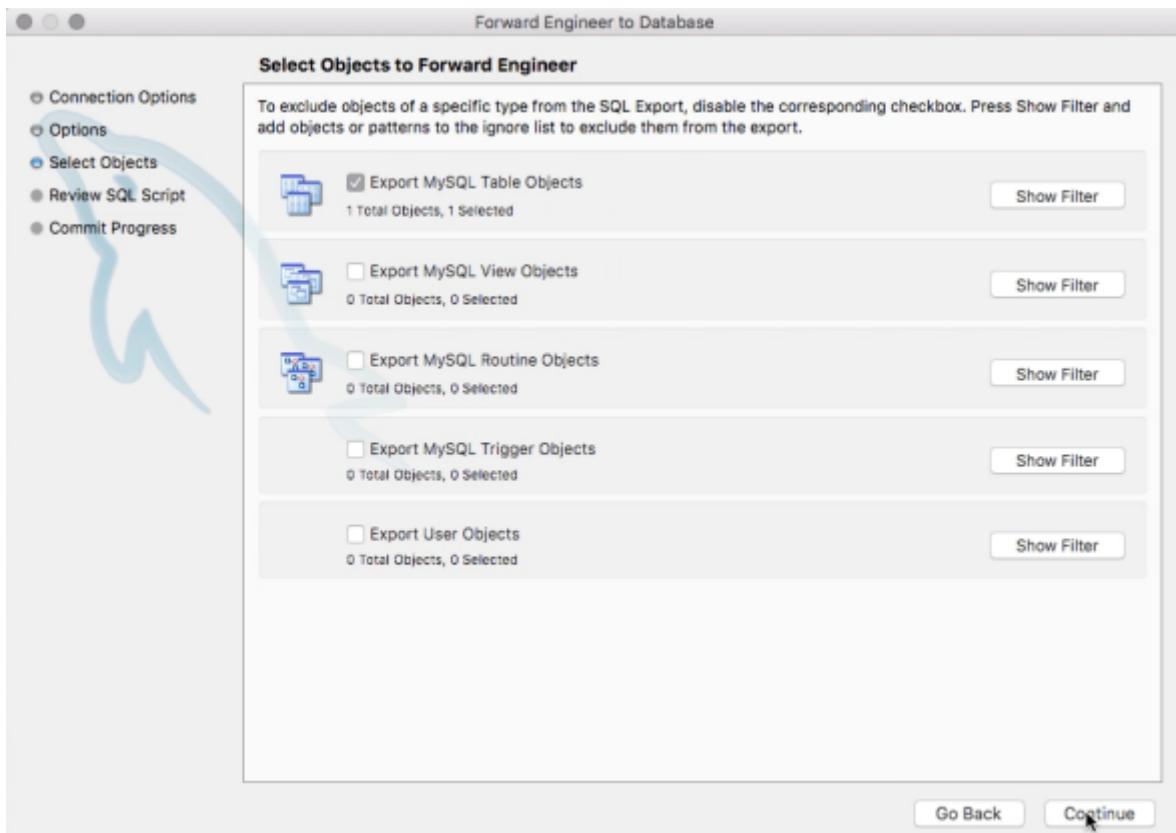


Figura 1.100: MySQL Workbench - Forward Engineering step 3

Verrà mostrato il codice SQL di creazione del modello prima di essere esportato sul server:

Forward Engineer to Database

Review the SQL Script to be Executed

This script will now be executed on the DB server to create your databases.
You may make changes before executing.

```
1 -- MySQL Workbench Forward Engineering
2
3 SET @OLD_UNIQUE_CHECKS=@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
4 SET @OLD_FOREIGN_KEY_CHECKS=@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
5 SET @OLD_SQL_MODE=@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';
6
7
8 -- Schema mydb
9
10
11
12 -- Schema mydb
13
14 CREATE SCHEMA IF NOT EXISTS `mydb` DEFAULT CHARACTER SET utf8 ;
15 USE `mydb` ;
16
17
18 -- Table mydb`.`person`
19
20 CREATE TABLE IF NOT EXISTS `mydb`.`person` (
21     `id` INT NOT NULL AUTO_INCREMENT,
22     `first_name` VARCHAR(45) NULL,
23     `last_name` VARCHAR(45) NULL,
24     `birth_date` DATE NULL,
25     `picture` BLOB NULL,
26     PRIMARY KEY (`id`)
27 ) ENGINE = InnoDB;
28
29
30 SET SQL_MODE=@OLD_SQL_MODE;
31 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
32 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

Save to File... Copy to Clipboard

Go Back Continue

Figura 1.101: MySQL Workbench - Forward Engineering step 4

Infine si esegue il salvataggio:

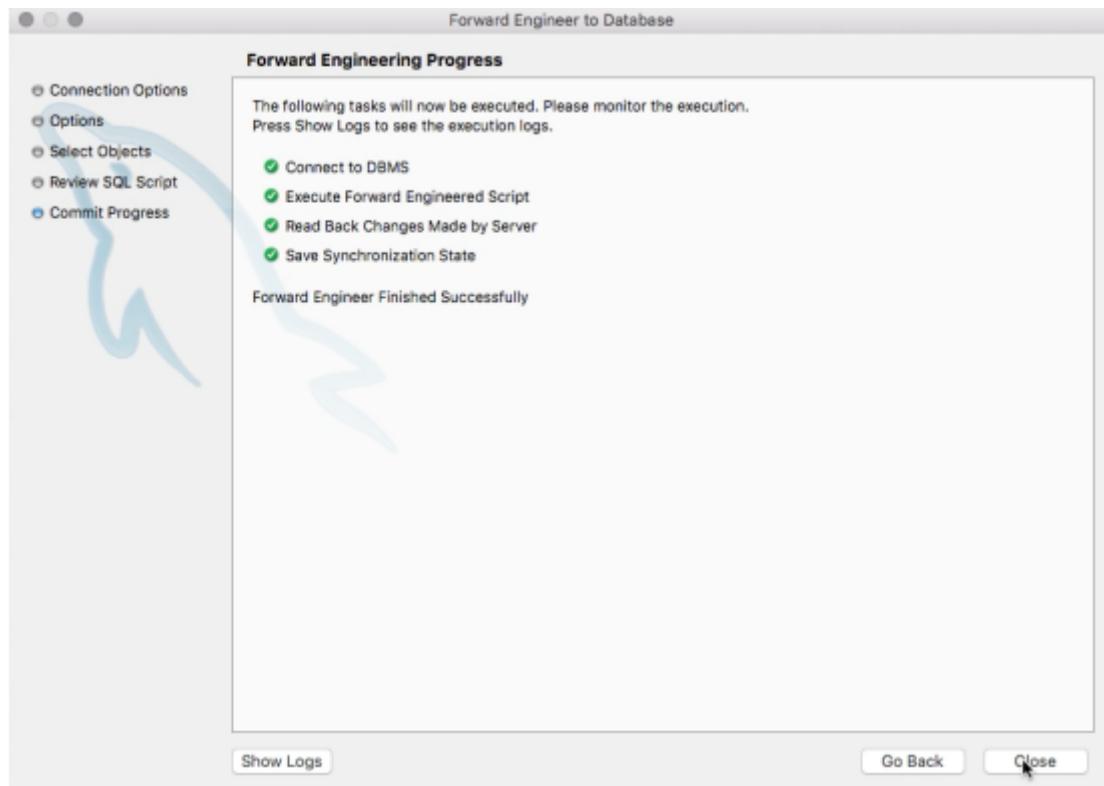


Figura 1.102: MySQL Workbench - Forward Engineering - Save

Andando ora a rivedere il contenuto del server (sempre tramite il Workbench) troveremo un nuovo database chiamato **mydb**:

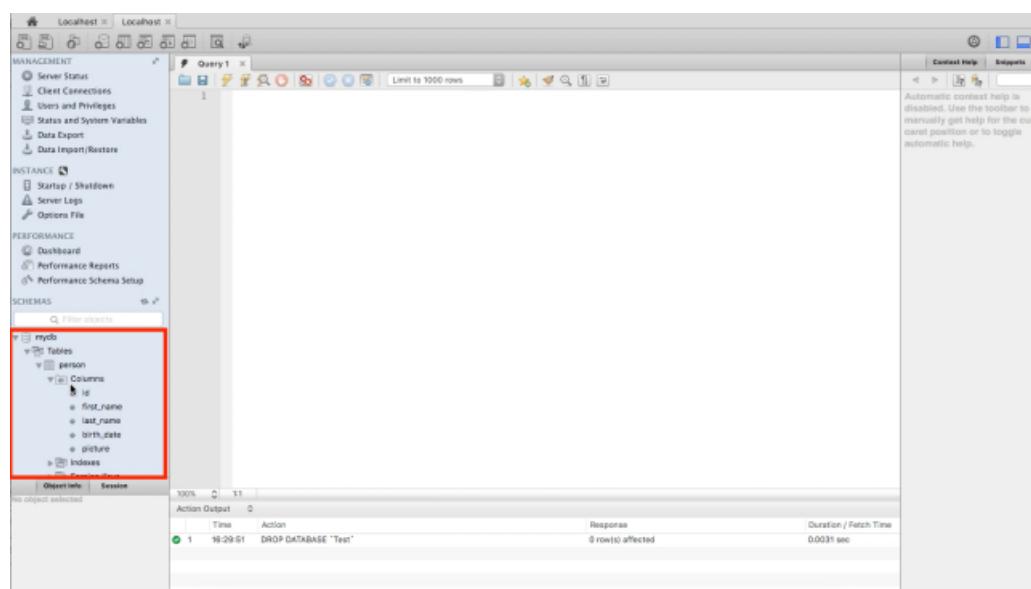


Figura 1.103: MySQL Workbench - mydb RECAP

cliccando col tasto destro sulla tabella **person** e selezionando: “Select Row – Limit 1000 ” verrà visualizzato il contenuto della tabella caricata sul server con la seguente schermata:

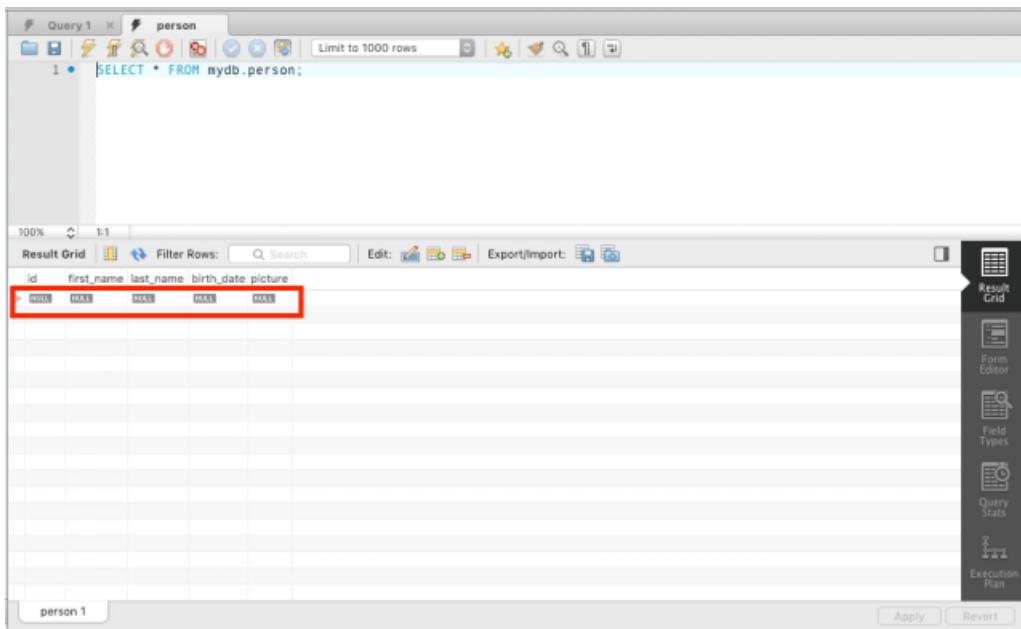


Figura 1.104: MySQL Workbench - person set on server

dove nella parte in alto viene visualizzata la “QUERY”, e in basso il risultato (ovviamente ora viene visualizzata una tabella vuota). A questo punto fare doppio click nella selezione rossa nell’immagine, in corrispondenza di `first_name`, e inserire il nome. Continuare così per il resto dei campi e poi premere invio, in questo modo si inseriranno i dati all’interno della tabella. Bisognerà ottenere un risultato come il seguente:

<code>id</code>	<code>first_name</code>	<code>last_name</code>	<code>birth_date</code>	<code>picture</code>
NULL	Mario	Rossi	2001/01/...	NULL
NULL	Giorgio	Bianchi	2002/02/...	NULL
NULL	Giuseppe	Verdi	2003/03/03	NULL
NULL	NULL	NULL	NULL	NULL

Figura 1.105: MySQL Workbench - Insert new people

In seguito in basso a sinistra si troverà il tasto “Apply”:

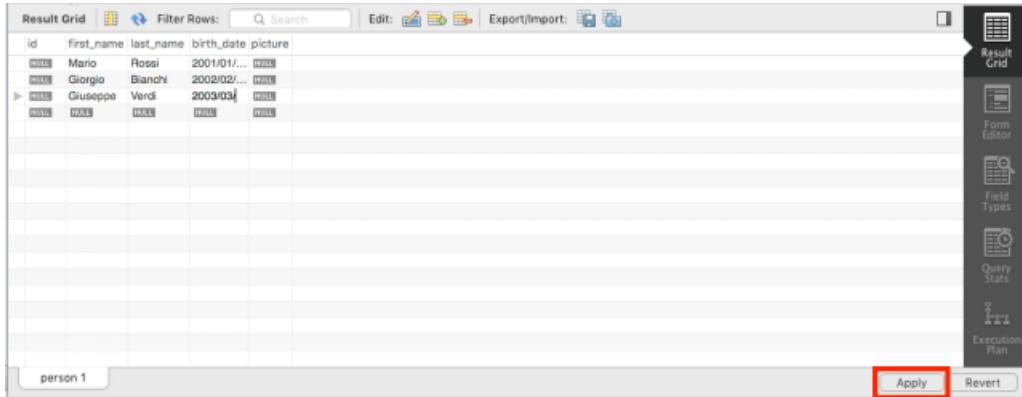


Figura 1.106: MySQL Workbench - Apply

Si aprirà il seguente wizard in cui sarà visualizzata la QUERY da eseguire sul database:

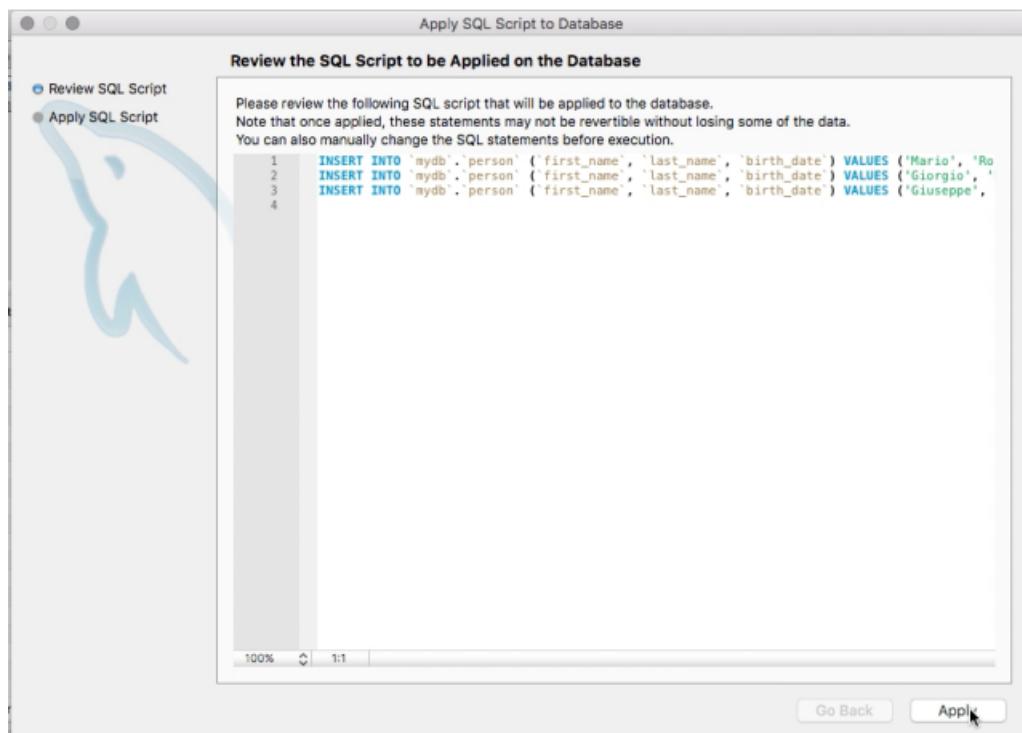


Figura 1.107: MySQL Workbench - Pre-query wizard

Cliccando su “Apply” verrà eseguita la query ottenendo il seguente risultato:

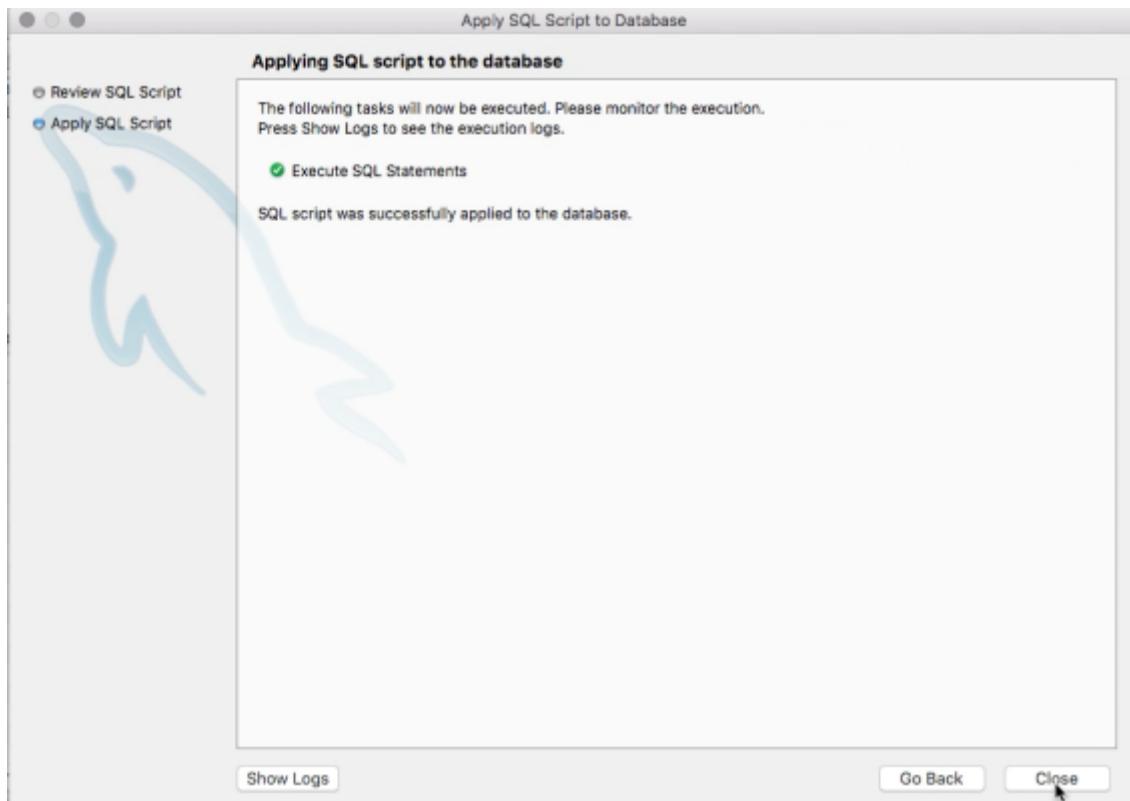


Figura 1.108: MySQL Workbench - Post-query wizard

Provare ora ad eseguire le QUERY:

- 1• **SELECT * FROM** mydb.person **WHERE** first_name = 'Mario'
- 1• **SELECT * FROM** mydb.person **WHERE** first_name = 'M'
- 1• **SELECT * FROM** mydb.person **WHERE** first_name = 'mario'

e proseguire facendo varie prove, modificando le QUERY.

Marco D'Amato
Marco Mameli
13/10/2016

1.10 IMPORTANZA MODELLI

1.10.1 IMPORTANZA MODELLO DATI

Tra i vari modelli dell'ingegneria del software, il modello dati ha una rilevanza particolare in quanto influisce in maniera determinante sulle funzionalità dell'applicazione. Tuttavia, i vari schemi di un sistema software sono:

- DATABASE MODEL: come già accennato, il più importante;

- BUSINESS MODEL: vagamente simile al diagramma dei casi d'uso, ha il compito di esplicare le ragioni presenti dietro lo sviluppo di un sistema software. Consiste nella creazione di scenari, nei quali i prodotti applicativi servono a creare delle soluzioni a problemi specifici. Questi schemi mostrano lo scambio di valore intorno all'applicazione (E3 VALUE);
- MODELLO DI SISTEMA: importante per la scalabilità e la realizzabilità dell'applicazione;
- PRESENTATION MODEL: in questo caso si parla di visibility design, dove i vari attori sono inseriti in un diagramma in cui sono riportate le varie funzioni relative ad ognuno di essi.



Figura 1.109: Customer Buys Product

Ultimamente quello dei disappearing computer è uno dei più interessanti ambiti di ricerca, dove per computer si intendono smart devices e smart processes: l'obiettivo è quello di rendere smart la parte relativa all'inserimento e al rilevamento dati;

- TEST: relativamente meno importante degli altri, ma validi da un punto di vista economico in quanto spesso è il collaudo a sbloccare i finanziamenti per il software.

1.10.2 IMPORTANZA INFORMAZIONE



Figura 1.110: Anthony's DMO Pyramid

Nella progettazione di un sistema software è importante progettare interfacce diverse per tipo di utente, creando diagrammi di comunicazione on the large per ogni utente.

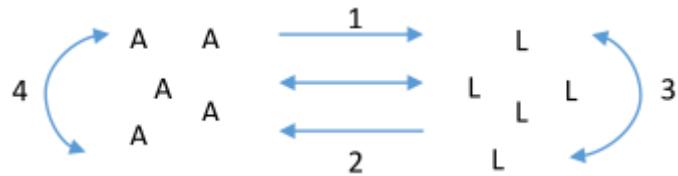


Figura 1.111: On the Large Communication Diagram

Esempio sui canali di comunicazione:

- Lettori leggono prodotto Autori;
- Autori leggono feedback dei Lettori;
- Lettori leggono feedback degli altri Lettori (casi generici Booking.com, TripAdvisor);
- Community di Autori.

1.11 DIAGRAMMI EER

Le tre caratteristiche della progettazione ad oggetti sono: Ereditarietà, Polimorfismo, Incapsulamento. Quest'ultimo non è presente nei Database, in quanto le entità non sono trasparenti; il polimorfismo è una componente essenziale per la programmazione, mentre non lo è per i dati. Per aggiungere l'Ereditarietà ai diagrammi E-R (Entity-Relationship) sono stati introdotti i diagrammi EER (Enhanced Entity Relationship).

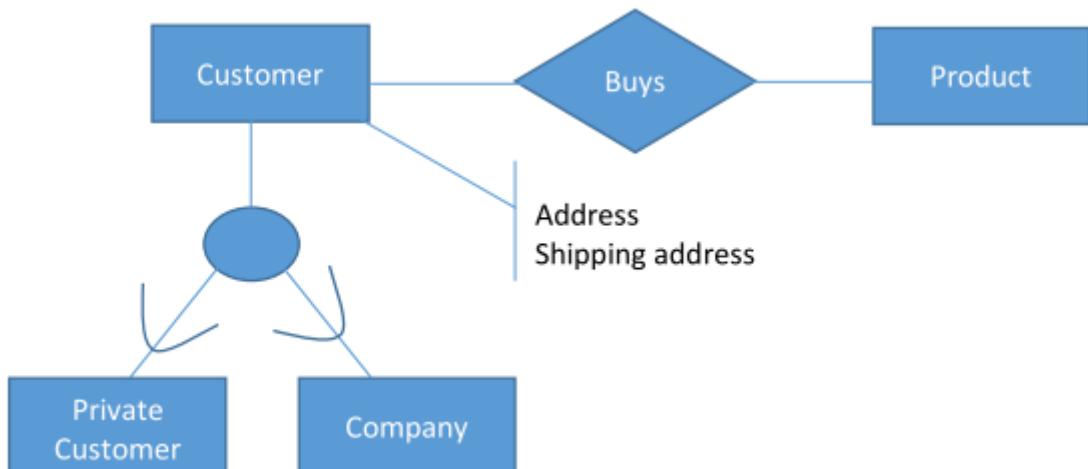


Figura 1.112: Diagramma EER

La modifica apportata allo schema di riferimento Cliente - Acquista - Prodotto rappresenta la specializzazione dell'entità cliente (si noti il simbolo di inclusione insiemistica). A questo punto si presentano due possibili situazioni:

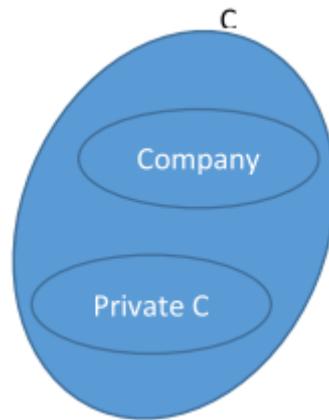


Figura 1.113: Diagramma di Eulero-Venn per EER disjoint entities

•

Le due entità pur essendo incluse nell'entità parente, non hanno elementi in comune. In questo caso si dicono disgiunte, e nel cerchio presente nel simbolo di specializzazione del diagramma EER si inserisce la lettera "d" (disjoint):

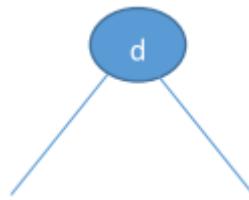


Figura 1.114: Disjoint Inheritance for EER

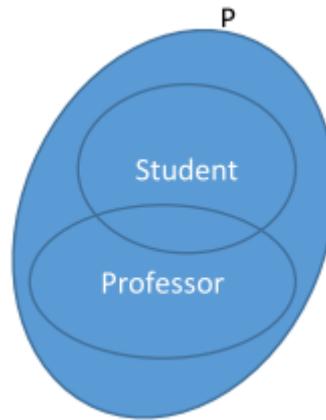


Figura 1.115: Diagramma di Eulero-Venn per EER overlapping entities

•

Le due entità potrebbero avere elementi in comune, come nel caso sotto riportato. In tal caso si inserisce la lettera "o" (overlap):

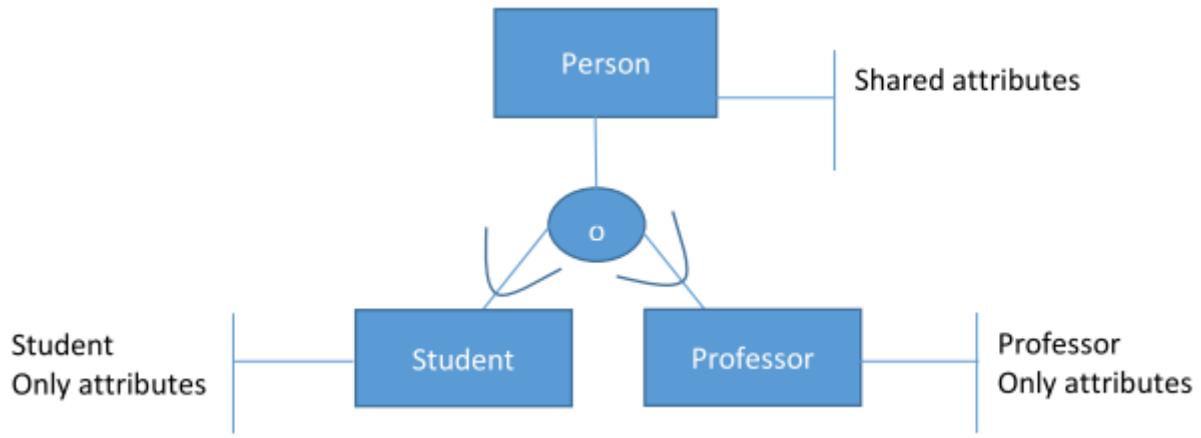


Figura 1.116: Overlapping Inheritance for EER

Esiste un importante differenza tra specializzazione e generalizzazione: La prima, effettuata a priori è sempre possibile, mentre la seconda non sempre. Nell'ultimo caso si può scoprire a posteriori che oggetti diversi appartengono alla stessa classe rendendo difficile per il software aggiornare la struttura dati. Un modo fattibile di implementare la generalizzazione è attraverso la UNION. Tale elemento di modellazione concettuale è rappresentato come sotto, dove in tal caso si inserisce il simbolo di “u” (union):

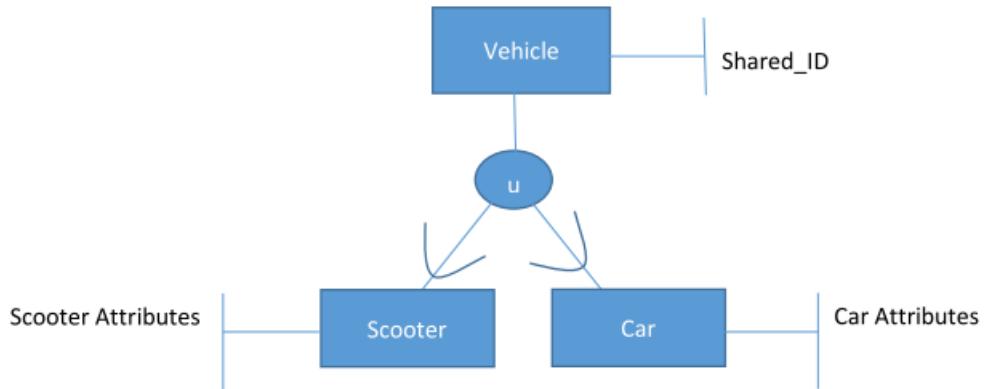


Figura 1.117: Union for EER

A differenza dell'Overlap, in questo caso le entità child mantengono i loro attributi ma sono unite sotto l'entità parent risolvendo parzialmente il problema: infatti nel caso della union si ha che fare con insiemi di oggetti diversi aventi attributi diversi. Si noti che l'entità Vehicle non contiene niente tranne che l'ID che però non è presente a livello concettuale ma solo a livello logico (dove sono presenti le tabelle).

1.11.1 SCHEMA EER CAPITOLO 4

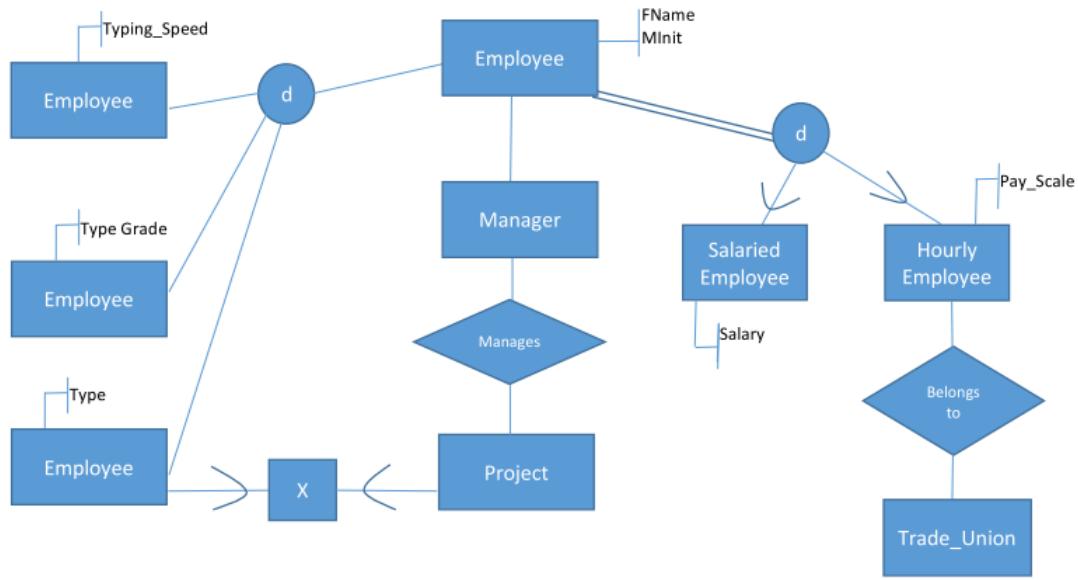


Figura 1.118: SCHEMA EER CAPITOLO 4

Nel diagramma sopra riportato la doppia linea rappresenta una specializzazione esclusiva, a indicare una partecipazione totale. Le due diverse specializzazioni disgiunte sono entrambe ortogonali, in quanto le entità child sono indipendenti l'una dall'altra. Nel caso di ereditarietà multipla non sempre le speciali portano ad alberi aperti: come si evince dal precedente diagramma possono esserci maglie chiuse. Il caso peggiore di queste situazioni si presenta quando gli attributi vengono gestiti due volte, anche se si tratta di un evento abbastanza raro.

1.11.2 ESERCIZIO SVOLTO IN CLASSE

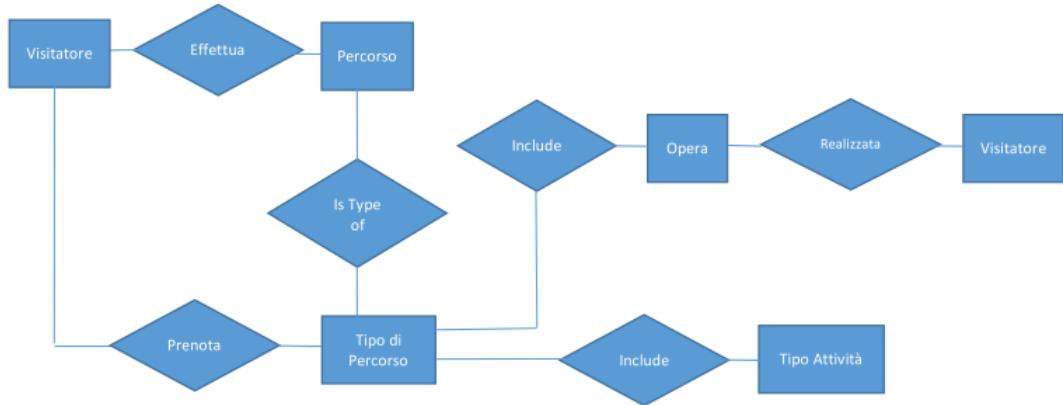


Figura 1.119: Museum Exercise ER

Simone Dongiovanni
19/10/2016

Capitolo 2

MODELLO LOGICO

2.1 SQL QUERY: Le interrogazioni del database

Per effettuare un'interrogazione del database bisogna avere una procedura sistematica che, dato un problema, ci consente di trovare una soluzione e verificarla in diversi casi di test in modo da appurare che sia una soluzione generale e non particolare.

Esempio: Una lista di clienti è una query sull'entità di tipo di cliente, ma non ci basta una sola query perché avremmo bisogno di vedere i clienti in ordine di nome, di luogo, di residenza, di data di nascita, ecc.



Figura 2.1: Cliente Acquista Prodotto

Le query devono rispondere ottimamente qualunque sia il criterio di filtraggio e devono essere robuste: ad esempio esiste la possibilità di, non conoscendo username e password, ma inserendo righe di codice in modo opportuno, poter entrare in un database (SQL INJECTION). Bisogna quindi fare attenzione nel momento in cui si scrive una query perché qualcuno potrebbe iniettarvi dei pezzi aggiuntivi. La connessione tra le pagine e il database avviene attraverso un certo numero di query: questo codice deve essere robusto, verificato, sviluppato attraverso procedure precise e rigorose basate sul concetto dell'algebra relazionale.

Le query sono formule matematiche scritte in algebra relazionale. Il modo migliore per scrivere una query è capire l'algebra che c'è dietro.

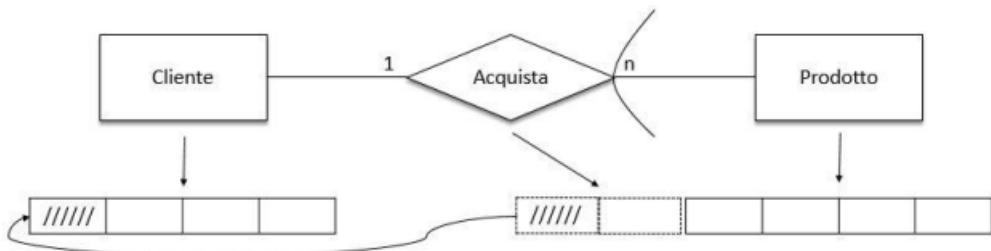


Figura 2.2: Cliente Acquista Prodotto - Mapping

Uno schema viene trasformato in tabella in base al tipo di cardinalità della relazione. Queste tabelle d'ora in poi saranno chiamate relazioni. L'algebra relazionale è un'algebra insiemistica in cui gli insiemi sono rappresentati non da diagrammi di Venn, ma da tabelle, in questo senso si dice che un database è un insieme di relazioni (R_i) e vincoli o constraints (C_j):

$$DB = \{R_i, C_j\}$$

I vincoli sono una serie di regole che utilizzeremo per far sì che i nostri dati restino coerenti, cioè che il database resti un database. Le caratteristiche principali sono: assenza di ridondanza, minima quantità di NULL, unicità di rappresentazione dell'informazione, ecc. L'oggetto di base dell'algebra relazionale, come detto, è la relazione o tabella. Oltre alle relazioni abbiamo un certo numero di operatori: come nell'algebra ci sono i numeri $\{1, 2, 3, \dots\}$ e li combiniamo attraverso operatori $\{+, *, \dots\}$, così in quella relazionale esistono operatori unari e binari. Un ulteriore modello è quello dell'algebra booleana dove esistono operatori unari e binari (ad esempio l'operatore *not* applicato ad un bit 0 fornisce come risultato 1, mentre *and* applicato a 1 fornisce 0).

2.1.1 ALGEBRA RELAZIONALE

Tutte le tabelle che vengono fuori dall'algebra relazionale sono calcolate, cioè vengono create come il risultato di un'espressione, pertanto sono solo viste, non vengono memorizzate e non occupano spazio.

- **Operatori Unari:**

Per quanto riguarda gli operatori unari dell'algebra relazionale:

- Il primo tra tutti gli operatori che ci interessano è quello di “*select*”. Si scrive come:

$$\sigma_{<c>}(R)$$

Questo operatore estrae o seleziona dalla tabella R una nuova tabella fatta dalle stesse colonne della tabella R e dalle sole righe che soddisfano la condizione C (fare riferimento cap. 4 e cap. 8 del libro).

Esempio Seleziona $\sigma_{<CLIENT.CITY='LECCE'>}(CLIENT)$. Cosa viene fuori da questa espressione? Avremo una nuova tabella chiamata R avente le stesse colonne della tabella clienti e un sottoinsieme delle righe;

- Il secondo operatore è “*project*”, così come la select riduce il numero di righe, la project riduce il numero di colonne estraendone solo alcune. Si scrive come:

$$\pi_{<col1, col2, \dots>}(R)$$

Questo operatore prende un numero selezionato o ridotto delle colonne della tabella di partenza;

- Il terzo operatore è “*rename*”, serve a rinominare determinati attributi. Questo produce una nuova tabella in cui le colonne si chiamano in un altro modo. Si scrive come:

$$\rho_{col1 \text{ as } new1, col2 \text{ as } new2, \dots}(R)$$

- **Operatori Binari:**

Il secondo gruppo di operatori sono di tipo insiemistico. Questi richiedono che gli oggetti siano union-compatible, cioè che abbiano le stesse colonne o che si chiamino nella stessa maniera. Le operazioni che conosciamo sono:

- Unione: denotata con $R \cup S$. Posso fare quest'operazione solo se la prima e la seconda tabella hanno le stesse colonne, ciò è garantito dalla union-compatible. Lo loro unione è una nuova tabella virtuale in cui ci sono tutte le righe della prima tabella e tutte quelle della seconda;
- Intersezione: denotata con $R \cap S$. Se ho due righe con gli stessi nomi di colonna, fare l'intersezione degli elementi presenti sia in R che in S significa selezionare gli elementi comuni. La chiave primaria deve garantire che ogni riga sia univocamente definita dato che la teoria dell'algebra relazionale dice che tutti i dati devono essere diversi;
- Differenza: denotata con $R - S = R - (R \cap S)$. È il set di tutti gli elementi che sono inclusi in R ma non in S;
- Prodotto cartesiano: $R \times S$. Questa operazione genera l'insieme di tutte le possibili coppie $T\{(r_i, s_j)\}$ Immaginiamo che R ed S siano dei concetti o idee, la capacità di creare corrispondenze tra queste genera dei nuovi concetti applicativi. Si intuisce, però, che l'insieme di tutte le coppie possibili è troppo grande (alcune non hanno senso e non servono), perciò eliminiamo le connessioni inutili utilizzando l'operazione di join;
- Join:

$$R \bowtie S := \sigma_{<c>}(R \times C)$$

Essa è l'operazione chiave dei database. È ciò che facciamo quando scriviamo *SELECT FROM WHERE*. La selezione è la possibilità di estrarre solo alcune colonne da una tabella potendo esprimere la condizione. In altre parole fare la join di due tabelle dal punto di vista dell'algebra relazionale significa attaccare tutti gli elementi corrispondenti.

Esempio: Join di due tabelle:

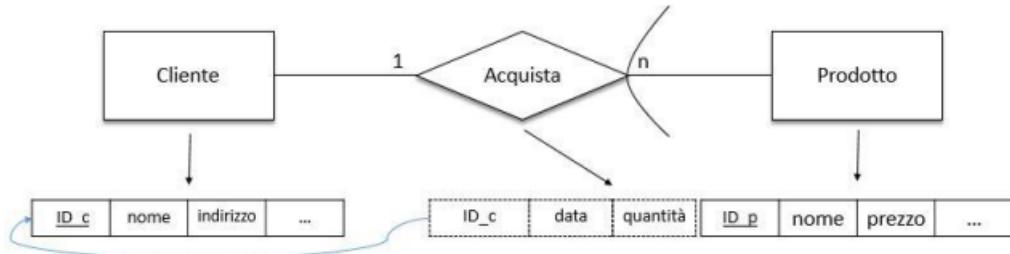


Figura 2.3: Join di due tabelle

Questa notazione senza righe è ciò che d'ora in poi chiameremo modello relazionale (insieme di tabelle più insieme di vincoli), in cui specifichiamo i vincoli di chiave primaria (sottolineata), di chiave esterna (ID_c) e vincoli di integrità referenziale (freccia).

2.1.2 Operatore Join

Analizziamo l'operatore join attraverso uno scenario: supponiamo di avere Mario Rossi con indirizzo Lecce e Giorgio Bianchi con indirizzo Milano, nel nostro autosalone abbiamo la macchina 1 che è una Fiat Tipo che ha il prezzo di € 10mila. Inizialmente nel DB non c'è scritto nulla ma nel momento in cui viene venduta ci aggiungiamo le informazioni riguardanti Mario Rossi con la data di acquisto. Fare il prodotto cartesiano delle due tabelle significa avere tutte le combinazioni o coppie possibili. Se adesso ci facciamo sopra una selezione sulla condizione di join, cioè vogliamo selezionare solo le righe $Prodotto.ID_c = Cliente.ID_c$ vedremo solo le coppie che soddisfano la corrispondenza.

Cliente				Prodotto						
ID_c	nome	indirizzo	...	ID_p	data	quantità	ID_c	nome	prezzo	...
1	Mario Rossi	Lecce			01/01/16	1	1	Fiat Tipo	10.000€	
2	Giorgio Bianchi	Milano								

Figura 2.4: Scenario logico Cliente/Prodotto

La join è una nuova tabella, che per il momento indichiamo come Cliente-Prodotto, contenente tutte le colonne del cliente $\{c_1, c_2, c_3, \dots\}$ e tutte le colonne del prodotto $\{p_1, p_2, p_3, \dots\}$ che soddisfano la condizione di join, ovvero che la chiave esterna di una tabella deve essere uguale alla corrispondente chiave primaria dell'altra tabella (se non ho venduto nessuna macchina la tabella di join sarà vuota).

Cliente - Prodotto								
ID_c	nome	indirizzo	data	quantità	ID_p	nome	prezzo	...
1	Mario Rossi	Lecce	01/01/16	1	1	Fiat Tipo	10.000€	

Figura 2.5: Cliente JOIN Prodotto

In algebra relazionale l'operazione di join è scritta in questo modo:

$$Cliente \bowtie_{<Prodotto.ID=Cliente.ID_c>} (Prodotto)$$

Unire le tabelle significa prendere le coppie degli elementi corrispondenti. Come si scrive questa cosa all'interno di una select? Possiamo ad esempio scrivere:

- 1 **SELECT** Cliente.nome, Prodotto.nome
- 2 **FROM** Cliente **JOIN** Prodotto **ON** Prodotto.ID_c = Cliente.ID_c
- 3 **WHERE** 1 — (significa prendili tutti)
- 4 — **WHERE** Prodotto.nome = "Fiat Tipo"

ove l'ultima *WHERE* commentata è un'ulteriore condizione per cercare tutti i clienti che hanno comperato una Fiat Tipo.

Vediamo ora la join dal punto di vista insiemistico:

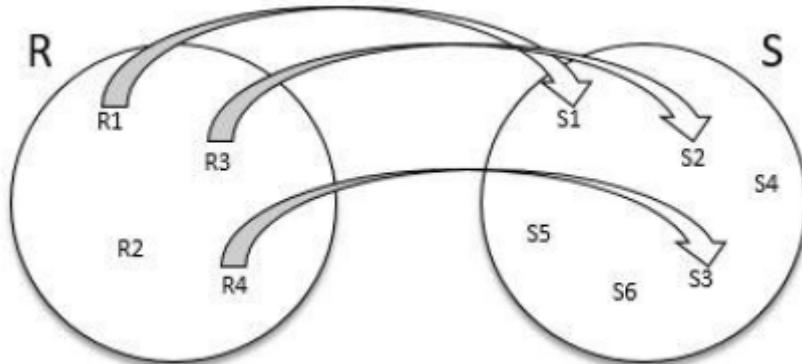


Figura 2.6: Diagramma Eulero-Venn per una JOIN

Tipi di Join

Grazie a questi concetti di base dell'algebra relazionale (più altri che aggiungeremo nelle lezioni successive) possiamo effettuare qualunque tipo di query. Tramite l'algebra relazionale se l'informazione è nel database possiamo estrarla grazie a questi operatori. Potremo aggiungere, inoltre, alcune varianti dell'operazione di join che non sono strettamente necessarie ma sono utili:

- Natural join $\{*\}$: è l'operazione che si fa quando all'interno di una condizione si specifica esattamente cosa è scritto nel vincolo di integrità referenziale, cioè è la join tra due tabelle quando esiste solo una colonna con lo stesso nome;
- Theta join $\{\theta\}$: è l'operazione che generalizza il concetto di operatore relazionale, cioè quelli che mettono in relazione due termini $\{<, \leq, ==, \geq, >\}$. Quest'operazione non si limita alla join (o equi-join) che verifica solo l'uguaglianza tra gli elementi, ma usa un operatore relazionale diverso per ottenere criteri di ordinamento più ampi (mette in confronto due tabelle per vedere tutti gli elementi a seconda della particolare relazione considerata);

Ulteriori varianti interessanti dell'operatore join, dal punto di vista insiemistico, sono:

- Left outer join $\{=\bowtie\}$: è l'operazione che prende tutti gli elementi dell'insieme R e solo i corrispondenti dell'insieme S;
- Right outer join $\{\bowtie=\}$: è l'operazione che prende tutti gli elementi dell'insieme S e solo i corrispondenti dell'insieme R;
- Full outer join $\{=\bowtie=\}$: è l'operazione che prende tutti gli elementi di entrambi gli insiemi, se ci sono delle corrispondenze le mette sulla stessa riga, altrimenti troveremo un NULL (a destra o a sinistra, mai da entrambe le parti).

Ricapitolazione:

- Select considera solo alcune righe;
- Project considera solo alcune colonne;
- Unione, intersezione e differenza sono i medesimi concetti elementari di teoria degli insiemi;
- Join mette in relazioni oggetti che stanno in tabelle diverse, crea una nuova tabella con tutte le colonne di entrambe le tabelle iniziali ma solo le righe dove è presente una corrispondenza. Questa è l'operazione principe dell'algebra relazionale.

Giuseppe D'Amuri
Federico De Luca
20/10/2016

2.2 Sviluppo di una Web Application

Lo scopo di questa lezione è quello di introdurre gli strumenti necessari allo sviluppo di una Web Application per l'analisi di un possibile scenario.

2.2.1 XAMPP

Per poter sviluppare delle applicazioni web si sfruttano delle piattaforme software che si differenziano tra loro in base alle componenti di base che le caratterizzano:

- sistema operativo;
- web server;
- DBMS;
- linguaggi di sviluppo per la programmazione web.

Durante l'esercitazione è stata usata XAMPP, una piattaforma software multipiattaforma e libera caratterizzata da un approccio user friendly. Le componenti di base che la caratterizzano sono:

- web server: Apache HTTP Server;
- DBMS (o database server): MySQL e MariaDB;
- server FTP: ProFTPD;
- mail server: Mercury Mail Transport System (solo per utenti Microsoft);
- linguaggi di programmazione: Perl, PHP e/o Python.

Il nome dell'applicazione XAMPP è un acronimo che sta ad indicare i software che la compongono (Apache, MariaDB, PHP e Perl); la “x” iniziale sta per “*x-cross platform*” e indica che il software è multipiattaforma. Inoltre è possibile scaricare le seguenti versioni create ad hoc per i diversi sistemi operativi:

- WAMPP (Windows);
- MAMPP (MacOs);

- LAMPP (Linux).

Una volta completata la procedura d'installazione si accede al pannello di controllo di XAMPP, il quale permette l'avvio dei vari server semplicemente con un click.

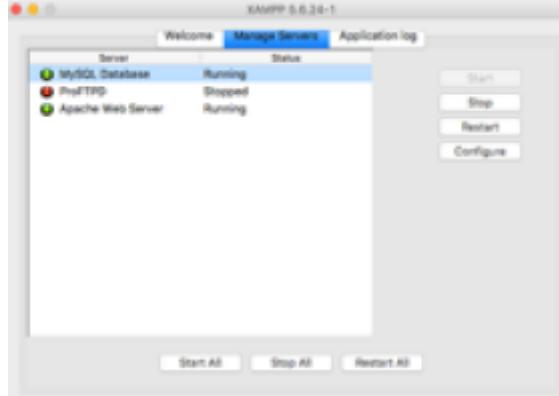


Figura 2.7: Pannello di controllo XAMPP

Avviati i server di Apache e MySQL (gli utenti Windows dovranno avviare anche Tomcat), è sufficiente collegarsi alla pagina: <http://localhost/dashboard/> e accedere alla pagina relativa alla voce ***phpMyAdmin*** per gestire il proprio database.

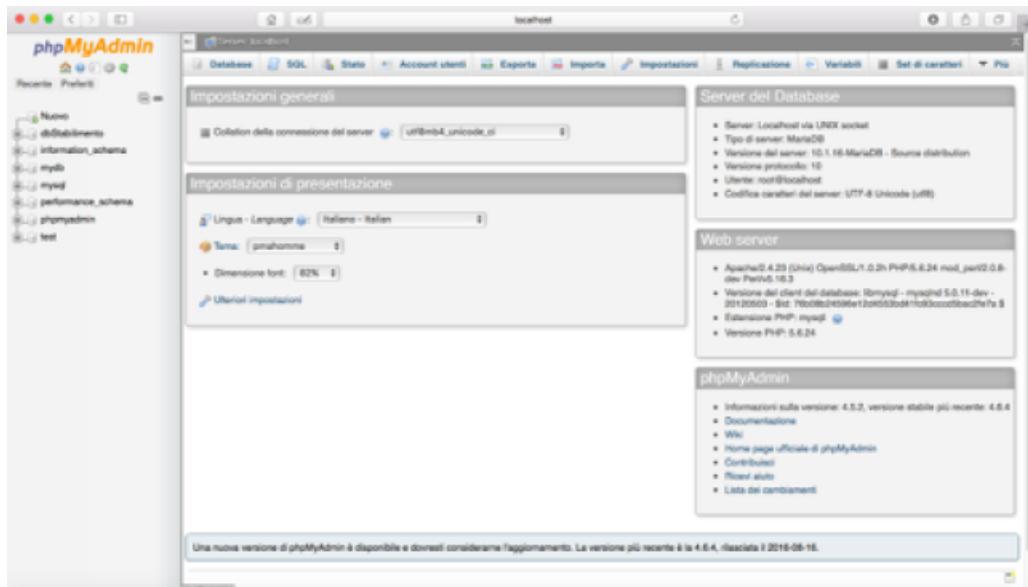


Figura 2.8: phpmyAdmin

Si prende in considerazione lo scenario in cui uno o più clienti acquistano uno o più prodotti.

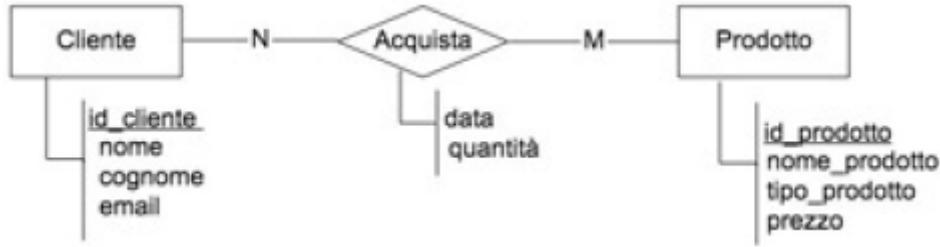


Figura 2.9: Cliente Acquista Prodotto

Il precedente modello ER presenta due entità (Cliente e Prodotto) legate dalla relazione Acquista (cardinalità N:M). Entità e relazione sono caratterizzate dalla presenza di alcuni attributi; per poter identificare in modo univoco le entità si useranno le chiavi primarie (id_cliente e id_prodotto). Su tale scenario si crea il database su **phpMyAdmin** direttamente cliccando sulla voce **Nuovo** nella pagina iniziale e andando a specificare il nome e la codifica caratteri del database (si consiglia **utf8.general_ci**, la quale permette l'uso di caratteri speciali ed è case insensitive). Fatto ciò vengono create le tabelle per le entità e la relazione. Ogni tabella sarà caratterizzata da un certo numero di colonne corrispondenti agli attributi, nelle quali si ha la possibilità di specificare informazioni quali il tipo, la lunghezza, la codifica e altro.

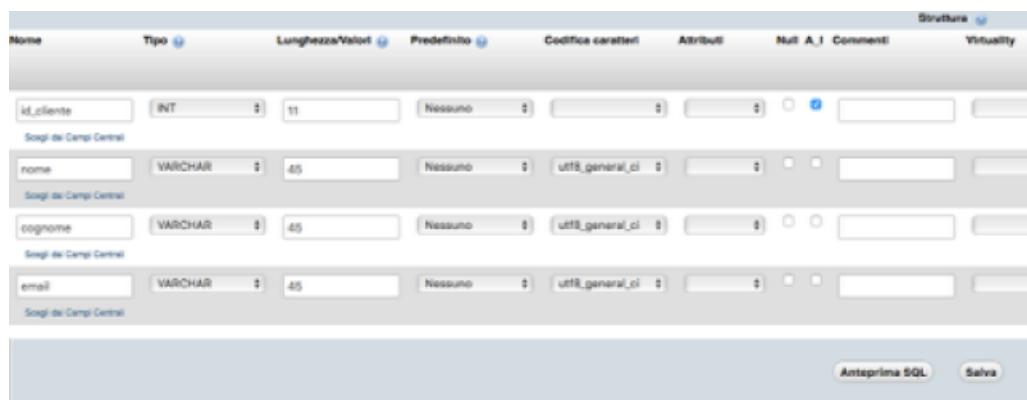


Figura 2.10: phpmyAdmin - Creazione tabella

Una di queste prevede la specifica della chiave primaria per identificare univocamente ogni record della tabella; per farlo bisogna inserire il valore **primary** nella sezione **Indice** relativa all'attributo che si vuole usare come chiave.



Figura 2.11: phpmyAdmin - Aggiungi indice

Diversamente dall'entità, la relazione è priva di chiavi primarie in quanto è univocamente identificata dalle chiavi esterne. Per specificare le chiavi esterne bisogna selezionare la tabella relativa alla relazione e cliccare sulla voce **Relazione Vista** per poi apportare le modifiche mostrate in figura indicando quale colonna usare come chiave esterna e in quale tabella è collocata.

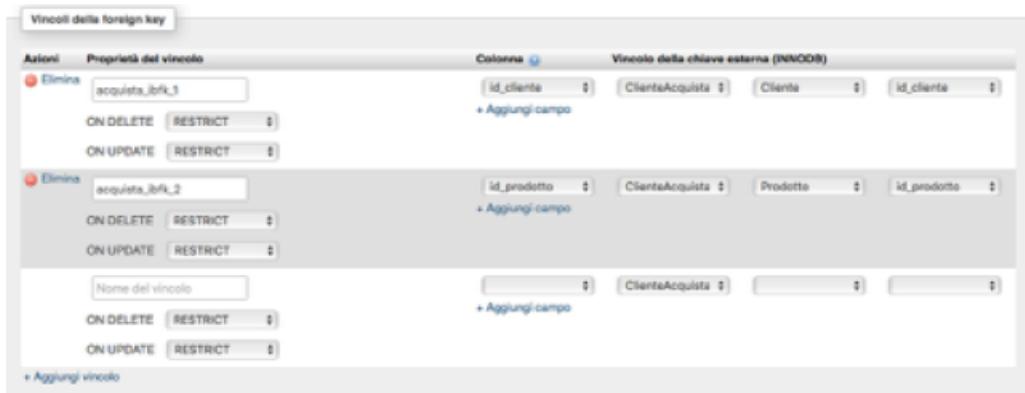


Figura 2.12: phpmyAdmin - Vincoli foreign key

Una volta completate, le tabelle possono essere riempite cliccando sulla voce **Inserisci** e compilando la seguente sezione.

Colonna	Tipo	Funzione	Null	Valore
id_cliente	int(11)		+	<input type="text"/>
id_prodotto	int(11)		+	<input type="text"/>
data	date		+	<input type="text"/>
quantità	int(11)		+	<input type="text"/>

Esegui

Ignora

Colonna	Tipo	Funzione	Null	Valore
id_cliente	int(11)		+	<input type="text"/>
id_prodotto	int(11)		+	<input type="text"/>
data	date		+	<input type="text"/>
quantità	int(11)		+	<input type="text"/>

Esegui

Figura 2.13: phpmyAdmin - Aggiunta record

Popolato il database, possiamo estrapolare il modello ER direttamente da *phpMyAdmin* cliccando sulla voce **Designer** nella barra degli strumenti dell'applicazione.

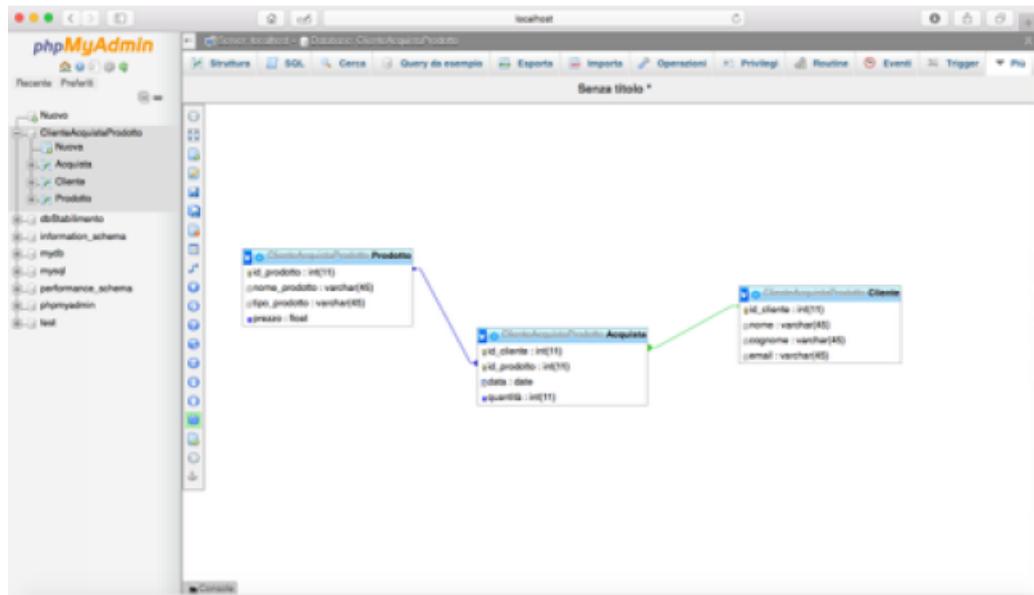


Figura 2.14: phpmyAdmin - Modalità designer

Si ha inoltre la possibilità di esportare tale modello in due modi:

- veloce, dove è possibile scegliere solo il formato (*.sql, *.csv, ...);
- personalizzato, dove oltre al formato, è possibile scegliere quali tabelle esportare, la codifica dei caratteri e altre opzioni aggiuntive.

Un'altra importante funzionalità di **phpMyAdmin** è la realizzazione di query tramite la schermata messa a disposizione dall'applicazione o per mezzo della **visual builder** che permette di costruire la query andando a scegliere graficamente gli attributi desiderati direttamente dal modello.

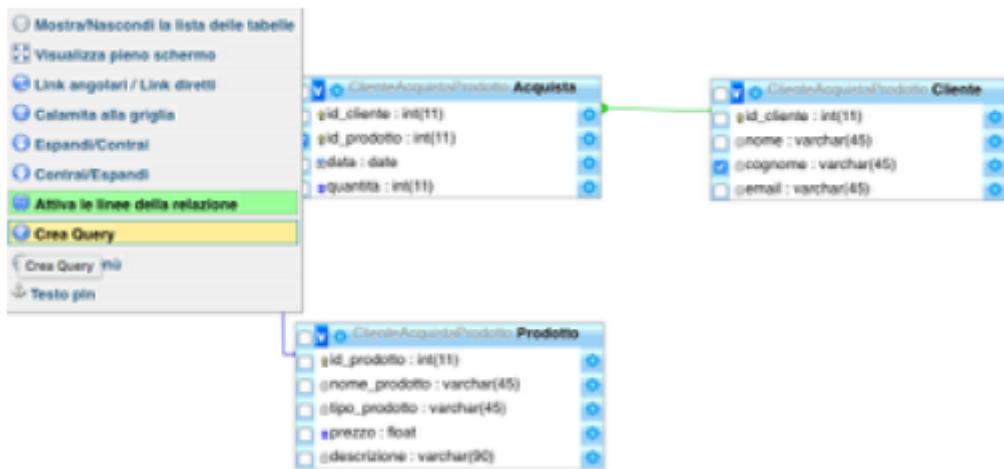


Figura 2.15: phpmyAdmin - Visual Builder

The screenshot shows the main phpMyAdmin interface. On the left, the database structure is visible with the 'Clienti-Acquista-Prodotti' schema selected. The central part of the screen contains a 'Query builder' section with fields for 'Colonna' (Column), 'Alias' (Alias), 'Mentre' (While), 'Ordinamento' (Ordering), 'Criterio' (Criteria), 'Inserisci' (Insert), 'Cancella' (Delete), and 'Modifica' (Modify). Below this is a toolbar with buttons for 'Aggiungi/Cancella criterio di riga' (Add/Delete row criteria), 'Aggiungi/Cancella campi' (Add/Delete fields), and 'Applica Query' (Apply Query). At the bottom, there is a 'Utilizza tabella' (Use table) button and a 'SQL-query sul database Clienti-Acquista-Prodotti' (SQL query on database Clienti-Acquista-Prodotti) text area containing the following SQL code:

```

1 SELECT `Cliente` . `nome` , `Acquista` . `data`
2 FROM `Prodotti` 
3 LEFT JOIN `Acquista` ON `Acquista` . `id_prodotto` = 
4 `Prodotti` . `id_prodotto` 
5 LEFT JOIN `Cliente` ON `Acquista` . `id_cliente` = 
6 `Cliente` . `id_cliente` 
7 WHERE `Prodotti` . `tipo_prodotto` LIKE '%tip%' 
8 ORDER BY `Cliente` . `nome` ASC

```

Figura 2.16: phpmyAdmin - Schermata query

L'applicazione presenta altre funzionalità quali la scelta dei privilegi da attribuire ai diversi account oppure la possibilità di apportare modifiche al database o anche l'opportunità di ottenere il dizionario dei dati in formato pdf.

Acquista

Colonna	Tipo	Null	Prefinito	Collegamenti a	Commenti	MIME
<i>id_cliente (Primaria)</i>	int(11)	No		Cliente -> id_cliente		
<i>id_prodotto (Primaria)</i>	int(11)	No		Prodotto -> id_prodotto		
data	date	No				
quantità	int(11)	No				

Indici

Chiave	Tipo	Unica	Compresso	Colonna	Cardinalità	Codifica caratteri	Null	Commenti
PRIMARY	BTREE	Sì	No	id_cliente	0	A	No	
				id_prodotto	0	A	No	
id_prodotto	BTREE	No	No	id_prodotto	0	A	No	

Cliente

Colonna	Tipo	Null	Prefinito	Collegamenti a	Commenti	MIME
<i>id_cliente (Primaria)</i>	int(11)	No				
nome	varchar(45)	No				
cognome	varchar(45)	No				
email	varchar(45)	No				

Indici

Chiave	Tipo	Unica	Compresso	Colonna	Cardinalità	Codifica caratteri	Null	Commenti
PRIMARY	BTREE	Sì	No	id_cliente	0	A	No	

Figura 2.17: phpmyAdmin - Ricapitolazione tabella

2.2.2 MySQL Workbench

MySQL Workbench è uno strumento visuale di progettazione per database che integra sviluppo SQL, gestione, modellazione dati, creazione e manutenzione di database MySQL all'interno di un unico ambiente. Il primo passo consiste nella creazione di una nuova connessione, specificandone il nome, il metodo di connessione e gli altri parametri richiesti.

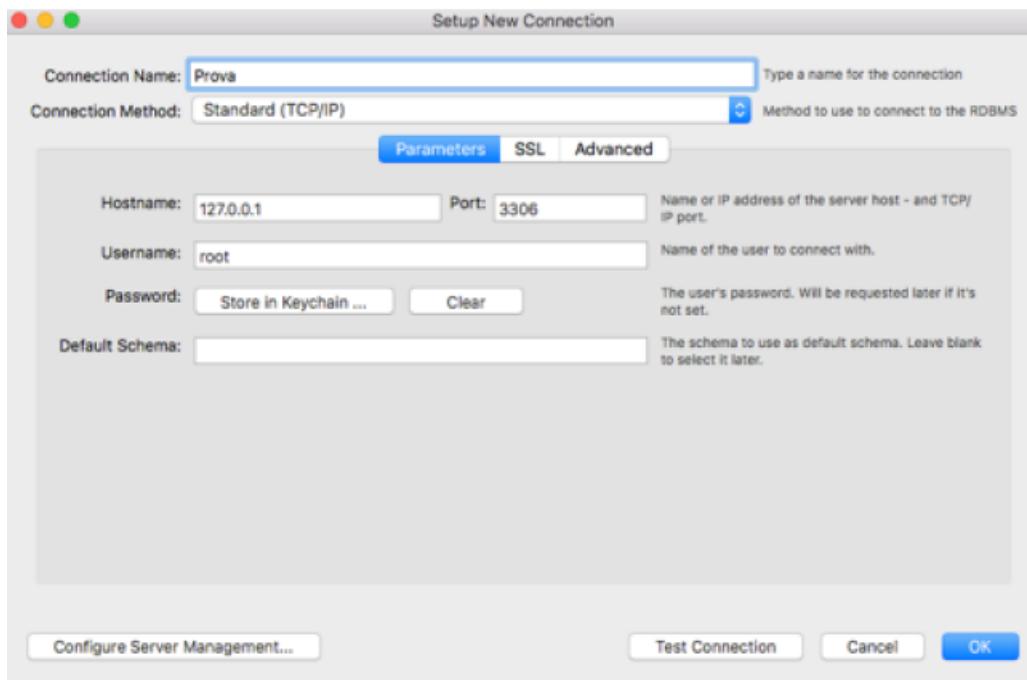


Figura 2.18: MySQL Workbench - New Connection

Creata e aperta la connessione si ha la possibilità di ottenere informazioni su:

- management;
- instance;
- performance;
- schemas.

Dalla sezione **management**, ad esempio, si può importare e/o esportare un database. Nella sezione **schemas** troveremo tutti gli schemi creati, tra cui quello realizzato precedentemente con **phpMyAdmin**. Selezionandolo è possibile gestire il proprio database, apportare delle modifiche, creare nuove tabelle o inserire dei record nelle tabelle presenti.

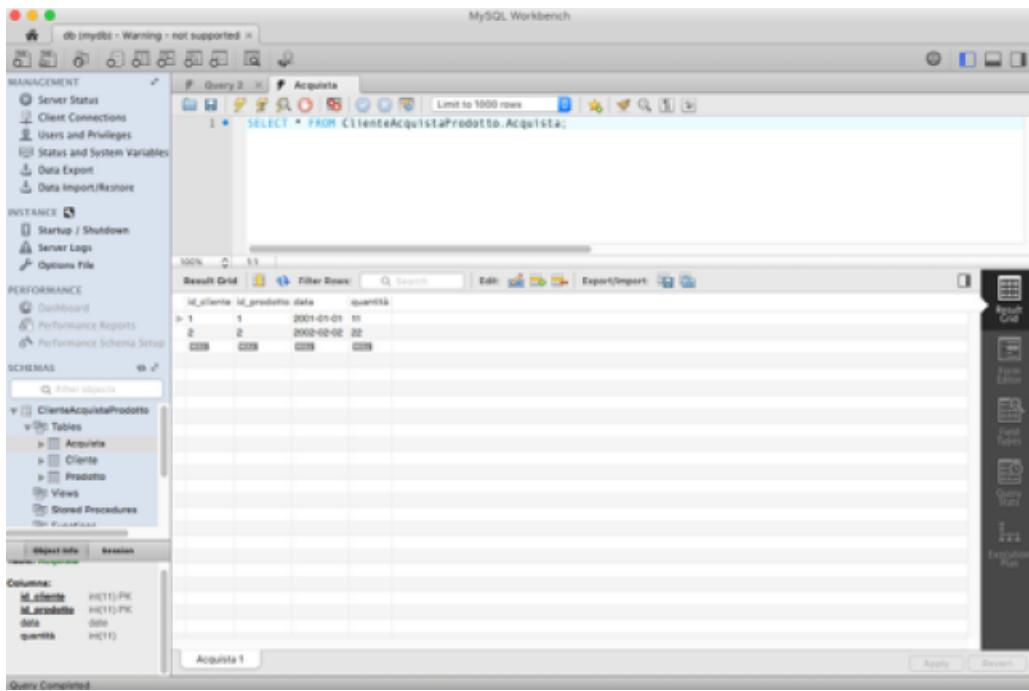


Figura 2.19: MySQL Workbench - Query

Una funzionalità importante consiste nel creare il diagramma EER a partire dallo schema; per farlo è sufficiente selezionare lo schema desiderato e cliccare sulla voce **Reverse Engineering** presente nel menu **Database**.

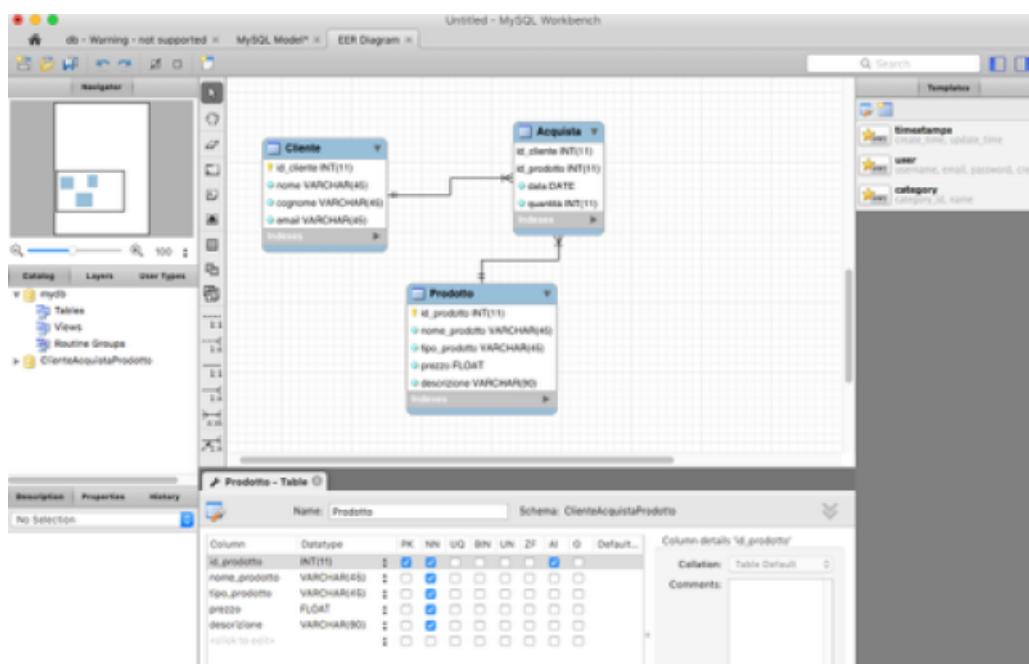


Figura 2.20: MySQL Workbench - Reverse Engineering

Ottenuto il diagramma è possibile apportare delle modifiche alle entità e agli attributi direttamente. Una volta modificato si ha la possibilità di sincronizzarlo con lo schema presente nel database tramite la voce **Synchronize Model** nel menu **Database**. Tramite questa funzionalità si modifica direttamente lo schema in modo da non avere diffidenze tra il diagramma EER e il database; l'applicazione notifica quale entità è stata modificata in modo da evitare errori e poi visualizza le query necessarie alle modifiche desiderate.

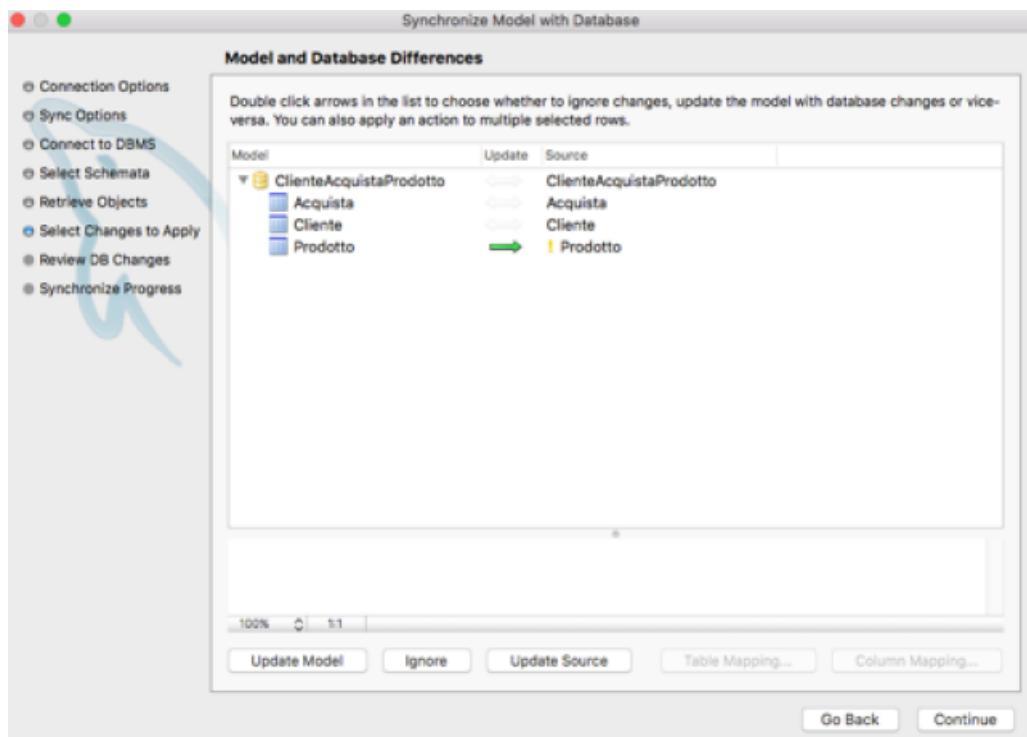


Figura 2.21: MySQL Workbench - Synchronize Model

2.2.3 DataGrip

DataGrip è un IDE sviluppato da JetBrains che permette di:

- accedere ai principali DBMS;
- modificare gli oggetti del database;
- scrivere in modo facilitato del codice SQL;
- eseguire query in modo facilitato.

Una delle caratteristiche interessanti, rispetto ai DBMS precedenti, è la possibilità del confronto tra una specifica sottoquery ed una tabella per vedere le eventuali righe aggiuntive rispetto ad una query selezionata. Diversamente dagli altri editor, DATAGRIP permette l'autocompletamento ed eventuali suggerimenti nella trascrizione dei comandi SQL. A parte alcune funzionalità aggiuntive, presenta le stesse caratteristiche già viste in phpMyAdmin e MySQL Workbench.

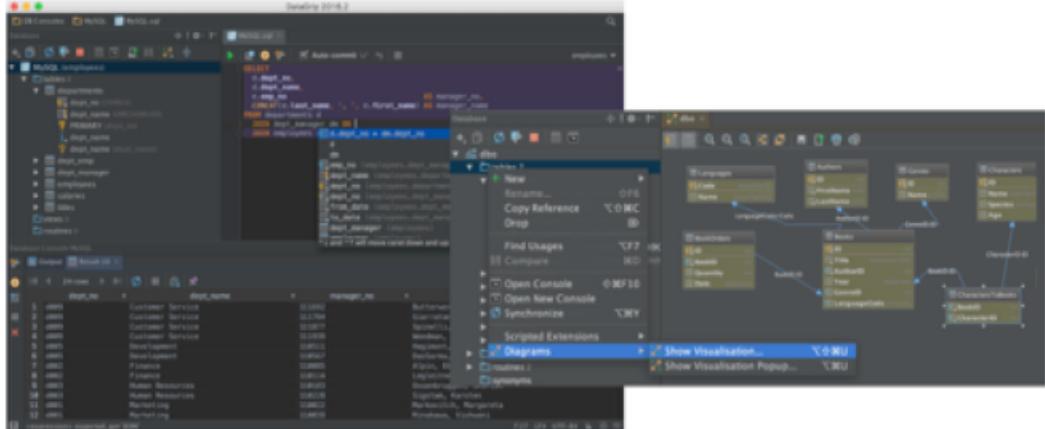


Figura 2.22: DataGrip

Luca Signore
Cristian Annicchiarico
20/10/2016

2.3 Algebra Relazionale

2.3.1 INTRODUZIONE

L’algebra relazionale è l’algebra su cui si basa il linguaggio SQL, cioè il linguaggio utilizzato per interrogare (to query) i databases. L’algebra relazionale opera su oggetti chiamati relazioni o tabelle, le quali sono sempre il risultato di un’espressione, quindi di un insieme di oggetti e operatori.

OPERATORI UNARI	NOTAZIONE	DESCRIZIONE
Select	$\sigma_{<c>}(\mathbf{R})$	Selezione delle righe di \mathbf{R} che soddisfano la condizione c
Project	$\pi_{<\text{col list}>}(\mathbf{R})$	Selezione di un numero ridotto di colonne di \mathbf{R} , quelle specificate in col list
Rename	$\rho_{<\text{col1 as new1, col2 as ...}>}(\mathbf{R})$	Serve a rinominare determinati attributi di \mathbf{R}

Figura 2.23: Operatori Unari Algebra Relazionale

Gli operatori binari, sono operatori che operano su insiemi, detti appunto *SET THEORETICAL*; operano su oggetti *Union-Compatible*, cioè su relazioni o tabelle con colonne dello stesso tipo (non necessariamente con colonne dello stesso nome).

OPERATORI SET THEORETICAL	NOTAZIONE	DESCRIZIONE
Unione	$R \cup S$	Restituisce una tabella contenente l'unione delle righe di R e di S
Intersezione	$R \cap S$	Restituisce una tabella contenente le righe comuni a R ed S
Differenza	$R - S$	Restituisce una tabella contenente le righe di R che non sono presenti in S
Prodotto Cartesiano	$R \times S$	Restituisce l'insieme di tutte le coppie possibili $T\{r_i, s_j\}$
Join	$R \bowtie_{<c>} S$ $= \sigma_{<c>}(R \times S)$	E' l'operazione di Select sul risultato del prodotto cartesiano, al soddisfacimento della condizione c Varianti dell'operatore Join sono (natural(*), theta (θ), left-right outer, full)

Figura 2.24: Operatori Binari Algebra Relazionale

Il grado di selettività di una Join è un parametro importante di cui bisogna tener conto per migliorare ad esempio l'efficienza di un motore di ricerca.

2.3.2 ALTRI OPERATORI

- **Le funzioni di aggregazione**

E' una famiglia di operatori utilizzati per effettuare analisi statistiche sui dati:

$< grouping\ attribute > F < function\ list > (R)$

Questo operatore, raggruppa i records secondo il criterio di raggruppamento specificato nel *grouping attribute* ed applica le funzioni specificate in *function list*. Possibili scelte di *function list* sono:

- **Count:** Conteggio;
- **Avg:** Media;
- **Sum:** Somma;
- **Max:** Valore massimo;
- **Min:** Valore minimo;
- **Var:** Varianza;
- **Std Dev:** Deviazione standard.

In linguaggio SQL:

```

1 SELECT function list
2 FROM ...
3 WHERE ...
4 GROUP BY grouping attribute
5 HAVING ... — condizioni sul risultato delle funzioni in function list

```

- **Operatore di Divisione**

Può essere considerato anch'esso un operatore di aggregazione, poiché, come per le funzioni di aggregazione, opera su più righe contemporaneamente.

R/S

Per spiegare bene come funziona, è utile osservare il seguente esempio:

Consideriamo R come la composizione di due gruppi distinti di campi (A, B), ed S come la composizione di uno solo dei due gruppi di R: $\Rightarrow R/S := T$:

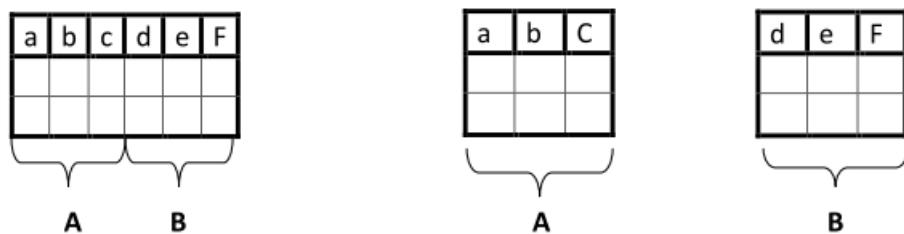


Figura 2.25: Operatore Divisione

In altre parole, come visto nell'esempio di prima, l'operatore diviso restituisce una tabella con solo le colonne di R che non sono in S .

L'insieme degli operatori visti finora è completo, poiché con esso si può interrogare il database in qualsiasi modo possibile, allo scopo di reperire tutte le informazioni di cui necessitiamo, se presenti. Il caso senz'altro più difficile da gestire è quello della chiusura ricorsiva, poiché gli operatori di cui abbiamo discusso consentono di interrogare il Database in tutti i modi finiti possibili, cioè solo nei casi in cui si ha una conoscenza a priori del numero di iterazioni di una data struttura. A titolo di esempio, supponiamo di dover creare un database contenente un albero genealogico:

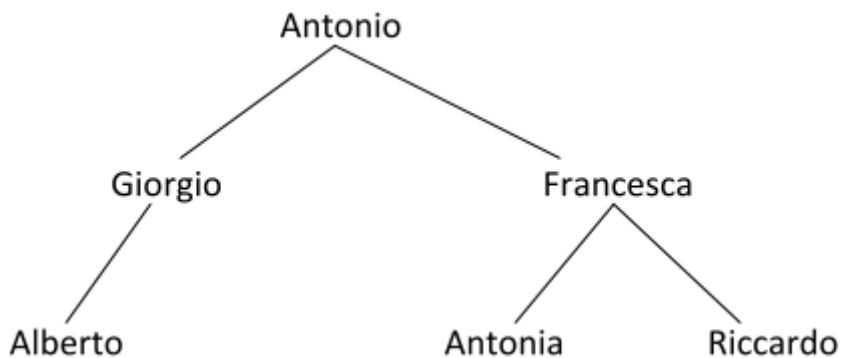


Figura 2.26: Albero genealogico

Il corrispondente modello ER e relativo modello relazionale, sono i seguenti:

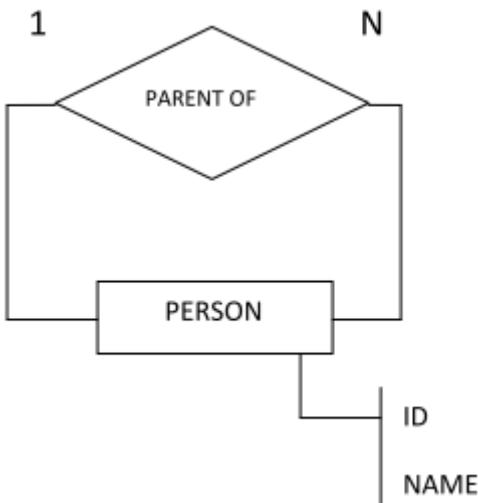


Figura 2.27: ER della relazione Parent Of

ID	NAME	PARENT_ID
1	Antonio	-
2	Giorgio	1
3	Francesca	1
4	Alberto	2
5	Antonia	3
6	Riccardo	3

Figura 2.28: Modello relazionale della relazione Parent Of

E' possibile vedere gli zii? Basta trovare il genitore di x e vedere i suoi fratelli. Applicando il prodotto cartesiano e poi applicando il vincolo di integrità referenziale nella join ($< PARENT_ID = ID >$), ottengo:

ID	NAME	PARENT_ID	ID	NAME	PARENT_ID
1	Antonio	-	2	Giorgio	1
2	Giorgio	1	4	Alberto	2
3	Francesca	1	5	Antonia	3
1	Antonio	-	3	Francesca	1
3	Francesca	1	6	Riccardo	3

GENITORE
FIGLIO
FIGLIO

Figura 2.29: Genitore JOIN Figlio

Successivamente, per controllare se il genitore di x ha fratelli, applico nuovamente l'operatore di Join. Questo esempio, spiega allora come non sia possibile trovare tutti i figli di un progenitore in “un colpo solo” a meno che non si conosca il numero di livelli di un albero.

2.3.3 ESEMPI DI QUERY

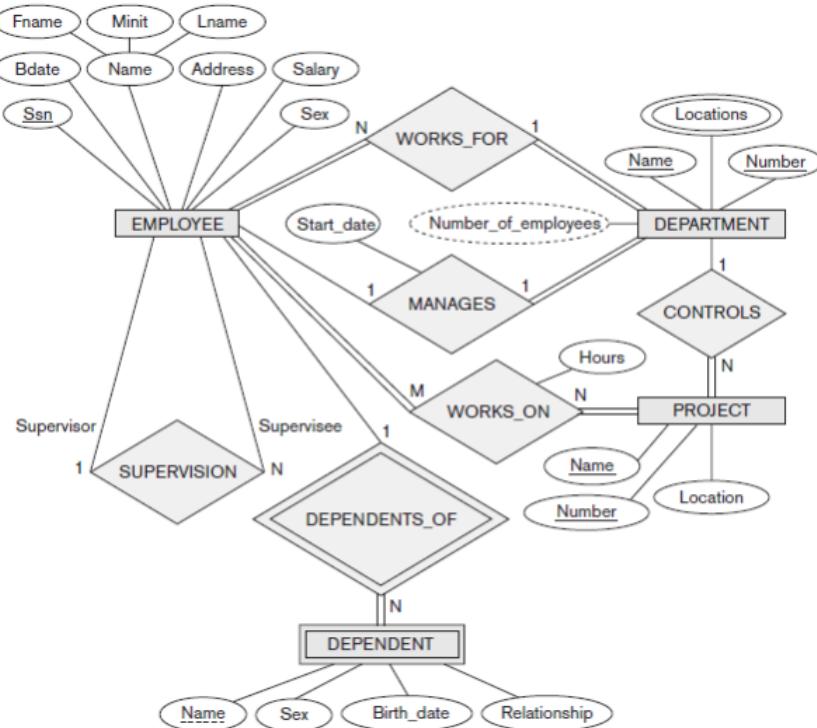


Figura 2.30: The Company Database

- **QUERY 1:** Nome e indirizzo di tutti gli impiegati che lavorano per il dipartimento di nome “Research”:

$$\pi_{<Name,Address>}(\sigma_{DEPARTMENT.Name='Research'}(DEPARTMENT \bowtie_{DEPARTMENT.number=EMPLOYEE.DEP_no} EMPLOYEE))$$

È necessario effettuare un'inner join tra Department ed Employee, selezionare con una select le righe in cui l'attributo Name vale "Research", e fare una project per ottenere soltanto le colonne Name ed Address.

n.b. sul testo la query è suddivisa in tre passaggi, per una maggiore chiarezza. Essi, però, non sono eseguiti in momenti distinti, ma in un unico blocco, come riportato sopra. La versione SQL della stessa query è:

```
1 SELECT Employee .Name, Employee .Address
2 FROM Employee JOIN Department ON Department .number=Employee .Dep_no
3 WHERE Department .Name=" Research"
```

- **QUERY 2:** *Per ogni progetto situato a Lecce, elencare il project number, il controlling department number e il nome del direttore che controlla il dipartimento.*

Le tabelle coinvolte sono Department, Employee e Project. Nel modello relazionale, il numero di dipartimento diventa attributo chiave esterna nella tabella Project (perché CONTROLS è una relazione 1:N nel modello ER). Assumiamo, inoltre, che la relazione Employee-MANAGES Department sia inclusa lato Department (l'ssn del direttore del dipartimento diventa attributo chiave esterna nella tabella Department). La query in SQL risulta essere la seguente:

```
1 SELECT Project .Number, Project .Dep_Number , Employee .Name
2 FROM ( Project JOIN Department ON Department .Number=Project .Dep_Number )
3                   JOIN Employee ON Department .employee_ssn=
                                Employee .Ssn
4 WHERE Project .Location=" Lecce"
```

Marco D'Amato
Adriano Luigi Piscopello
26/10/2016

2.4 Queries

L'insieme degli operatori fondamentali dell'algebra relazionale è composto da operatori associativi e non associativi, quest'ultimi agiscono su una singola entità, (selezione di determinate tuple, di particolari attributi ecc) mentre i primi operano su più entità diverse.

Le **operazioni associative** sono alla base della nostra stessa memoria, infatti queste hanno il compito di mettere insieme diverse entità, in modo da cercare degli elementi in base a dei criteri prestabiliti. Un esempio potrebbe essere quello di cercare tutti i frequentanti di un corso che hanno i capelli lunghi.

Gli **operatori associativi** li possiamo associare alle funzioni di aggregazione e la divisione.

Le **funzioni di aggregazione** permettono, come dice il nome stesso, di aggregare più informazioni in una sola, ad esempio possiamo vedere quanti impiegati ci sono in un'azienda, senza voler prendere il nome di ognuno. Questi operatori ci permettono di effettuare qualsiasi query sul database (possiamo estrarre qualsiasi informazione dal database, se presente, escluso il caso di recursive closure). Facciamo un esempio pratico, partendo da un database studiato precedentemente: Il "Company" Database.

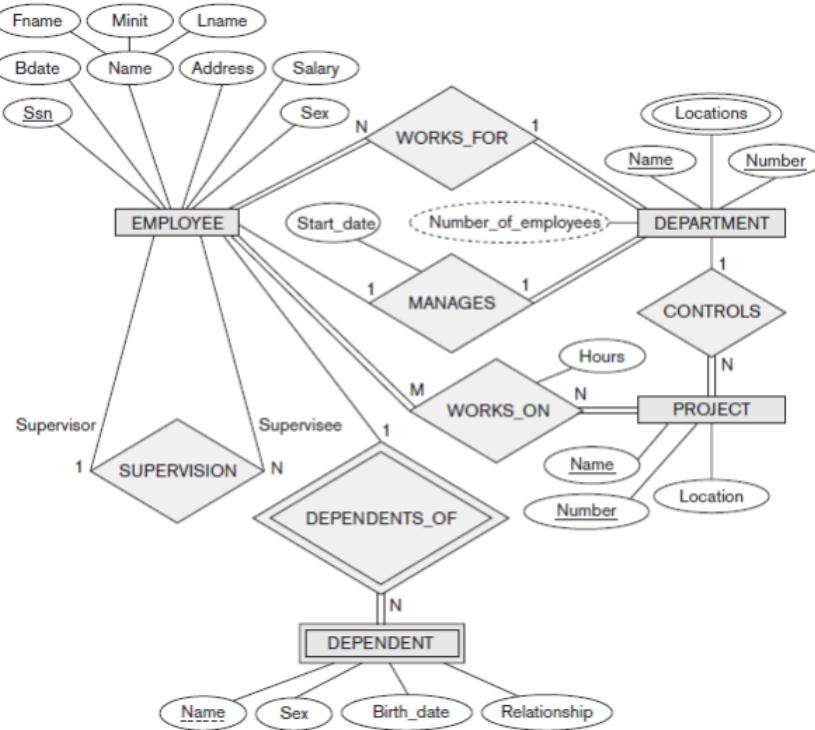


Figura 2.31: The Company Database

Attraverso le conoscenze pregresse, riusciamo a estrarre mentalmente quali saranno le tabelle che avremo sul database guardando direttamente il modello concettuale. L'entità *department* ha un attributo multiplo *locations*. Questo attributo, che nel modello logico diverrà una **weak entity type**, avrà una relazione *department has location* con cardinalità 1 a n. Questa trasformazione si può effettuare per qualsiasi attributo multiplo o composto del modello ER. Nella trasformazione tra modello ER a modello logico si ha un problema comune: Qual è la primary key? Nella tabella *department*, nel caso di diagramma ER, le primary keys saranno *name* e *number*. Queste a livello logico dovranno essere sostituite da una primary key numerica per ottimizzare le **join**, mentre, sempre a questo livello, gli attributi *name* e *number* dovranno essere inseriti come normali attributi imponendo, se necessario, le condizioni di **NOT NULL** e **UNIQUE**. Queste particolari trasformazioni permettono di sfruttare la **gestione dei sospesi**: possiamo inserire tutti i dati senza la chiave primaria concettuale, per poi rimanere in sospeso fino a quando non decidiamo di chiuderla e salvarla sul database.

Tornando al nostro esempio, dal diagramma ER possiamo vedere che la relazione *employee-manages-department* ha cardinalità 1 a 1, quindi possiamo mettere la chiave primaria di “*employee*” come chiave esterna a *department* (*director_id*). Per la relazione *employee-works-for-department*, possiamo notare che ha cardinalità n a 1, quindi dobbiamo inserire la chiave primaria di *department* come chiave esterna in *employee* (*department_id*). Come detto precedentemente, dobbiamo creare la tabella *location* e, avendo la relazione n a 1 con *department*, dobbiamo aggiungere la chiave esterna *department_id* alla tabella *location*. L'entità *project* diventa una tabella con id, tutti gli altri attributi e la chiave esterna al dipartimento associato (*department_id*). La relazione *employee-works-on-project* è n a m, quindi si crea una nuova tabella con le chiavi esterne (primarie) che puntano alle tabelle a cui è associata (*employee_id*, *project_id*) e gli attributi aggiuntivi (*hours*). Per la relazione *employee-supervisions-employee*,

aggiungiamo un attributo chiave esterna alla tabella *employee* (*supervisioner_id*) in riferimento alla tabella *employee* stessa.

Proviamo a risolvere i seguenti esercizi tramite l'utilizzo di queries. Si noti che l'utilizzo di più queries per la risoluzione di un singolo problema è puramente per motivi di spazio e leggibilità.

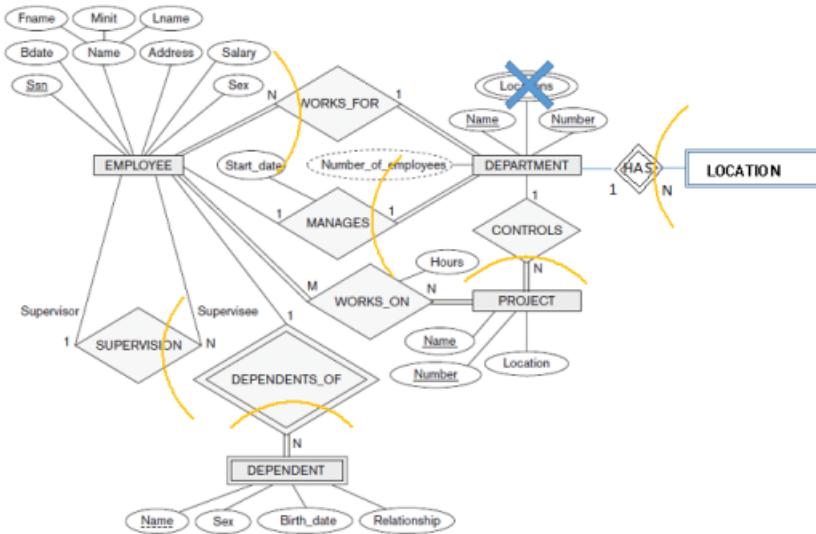


Figura 2.32: The Company Database - Logical Mapping Inclusions

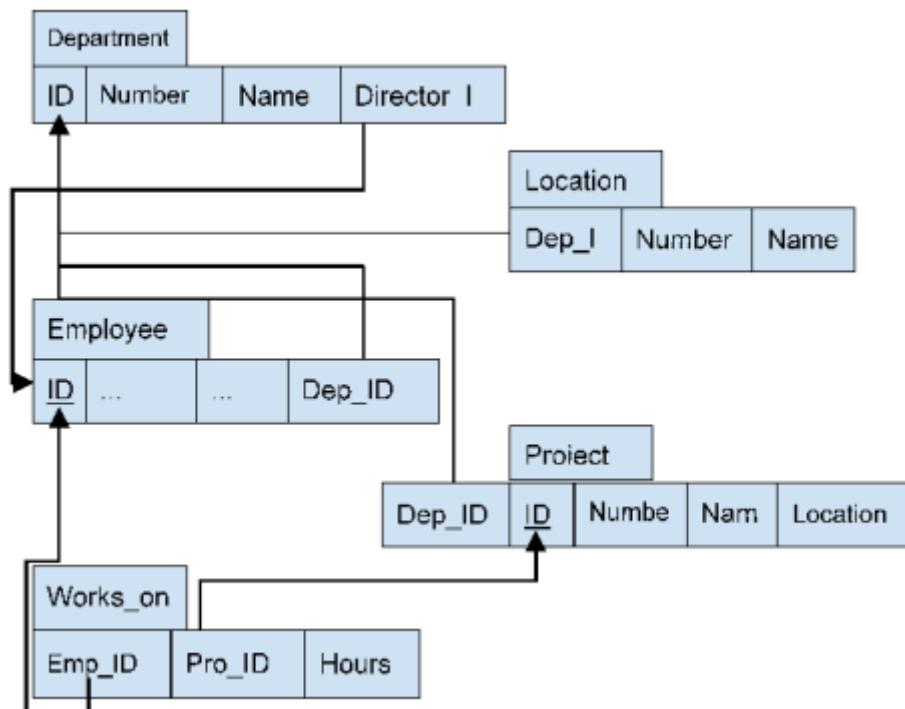


Figura 2.33: The Company Database - Logical

- **Query 1:** Restituire il nome e l'indirizzo di ogni impiegato che lavora per il dipartimento “ricerca”.

$$\pi_{\langle name, address \rangle} (employees \bowtie_{e.depId=d.id} \sigma_{\langle name = "research" \rangle} (department))$$

- Estraiamo tramite l’operazione di **select**, i dipartimenti che hanno nome “ricerca”;
- Il risultato ottenuto verrà unito, tramite comando **join**, alla tabella *employee* col vincolo di chiave esterna (*employee.dep_id* = *department.id*);
- A quest’ultimo risultato verrà applicata una **projection** sui campi *name* e *address*, per estrarre solo questi attributi.

- **Query 2:** Trovare i nomi degli impiegati che lavorano per tutti i progetti controllati dal dipartimento numero 5.

$$\left\{ \begin{array}{l} DEPT5_PROJS \leftarrow \rho_{(Pno)}(\pi_{\langle Pnumber \rangle}(\sigma_{\langle Dnum=5 \rangle}(project))) \\ EMP_PROJ \leftarrow \rho_{(Ssn, Pno)}(\pi_{\langle Essn, Pno \rangle}(works_on)) \\ RESULT_EMP_SSNS \leftarrow EMP_PROJ / DEPT5_PROJS \\ RESULT \leftarrow \pi_{\langle Lname, Fname \rangle}(RESULT_EMP_SSNS * employee) \end{array} \right.$$

- Estraiamo i progetti con numero di dipartimento = 5 e rinominiamo l’attributo *project_number* in *Pno*;
- Estraiamo l’ssn e il *project_number* dalla tabella *works_on*;
- Dividiamo il risultato della seconda query per il risultato della prima per avere l’ssn dei dipendenti che lavorano su TUTTI i progetti del dipartimento 5;
- Facciamo una **join** con la tabella *employee* per restituire nome e cognome dei dipendenti richiesti.

- **Query 4:** Creare una lista di progetti che hanno almeno un impiegato di cognome “Smith”, o come lavoratore o come manager del dipartimento che controlla il progetto.

$$\left\{ \begin{array}{l} SMITHS(Essn) \leftarrow \pi_{\langle Essn \rangle}(\sigma_{\langle Lname = "Smith" \rangle}(employee)) \\ SMITH_WORKER_PROJS \leftarrow \pi_{\langle Pno \rangle}(works_on * SMITHS) \\ MGRS \leftarrow \pi_{\langle Lname, Dnumber \rangle}(employee \bowtie_{ssn=mgrSsn} department) \\ SMITH_MANAGED_DEPTS(Dnum) \leftarrow \pi_{\langle Dnumber \rangle}(\sigma_{\langle Lname = "Smith" \rangle}(MGRS)) \\ \quad \left[\begin{array}{l} SMITH_MGR_PROJS(Pno) \leftarrow \\ \quad \left[\begin{array}{l} \leftarrow \pi_{\langle Pnumber \rangle}(SMITH_MANAGED_DEPTS * project) \end{array} \right] \end{array} \right] \\ RESULT \leftarrow (SMITH_WORKER_PROJS \cup SMITH_MGR_PROJS) \end{array} \right.$$

- Restituiamo il valore degli ssn dei dipendenti di cognome “Smith” tramite select e **projection**;
- Troviamo gli id dei progetti in cui lavorano impiegati di nome “Smith” (trovato dalla query precedente);
- Restituiamo il nome dei manager dei vari dipartimenti con il corrispettivo numero di dipartimento;

- Selezioniamo gli ssn degli impiegati col cognome “Smith” dalla query precedente;
- Troviamo il numero dei progetti associati ai dipartimenti restituiti dalla query precedente;
- Uniamo i risultati della seconda e della quinta query.

- **Query 5:** *Lista dei nomi di tutti gli impiegati con 2 o più parenti a carico.*

$$\pi_{<Lname, Fname>}(\sigma_{<count \geq 2>} (ssn F_{COUNT} name(dependent)) * employee)$$

- Contiamo il numero di familiari a carico presenti per ogni ssn degli impiegati (utilizzando l'apposita funzione di aggregazione **COUNT**);
- Selezioniamo tramite **select** gli ssn che hanno $count \geq 2$;
- Il risultato verrà unito tramite **join** alla tabella *employee* per restituire il nome e il cognome degli impiegati richiesti.

- **Query 6:** *Restituire i nomi dei dipendenti che non hanno parenti a carico.*

$$\left\{ \begin{array}{l} ALL_EMPS \leftarrow \pi_{Ssn}(employee) \\ EMPS_WITH_DEPS(Ssn) \leftarrow \pi_{Essl}(dependent) \\ EMPS_WITHOUT_DEPS(Ssn) \leftarrow (ALL_EMPS - EMPS_WITH_DEPS) \\ RESULT \leftarrow \pi_{Lname, Fname}(EMPS_WITHOUT_DEPS * employee) \end{array} \right.$$

- Estraiamo la lista di ssn degli impiegati;
- Estraiamo la lista di ssn degli impiegati con familiari a carico;
- Sottraiamo il primo risultato col secondo;
- Restituiamo, tramite **join** con la tabella *employee*, il nome e il cognome dei dipendenti rimasti.

- **Query 7:** *Restituire il nome dei manager che hanno un familiare a carico.*

$$\left\{ \begin{array}{l} MGRS(Ssn) \leftarrow \pi_{MgrSsn}(department) \\ EMPS_WITH_DEPS(Ssn) \leftarrow \pi_{Essl}(dependent) \\ MGRS_WITH_DEPS \leftarrow (MGRS \cap EMPS_WITH_DEPS) \\ RESULT \leftarrow \pi_{Lname, Fname}(MGRS_WITH_DEPS * employee) \end{array} \right.$$

- Estraiamo l'elenco degli ssn dei managers dei dipartimenti;
- Estraiamo l'elenco degli ssn degli impiegati con familiari a carico;
- Intersechiamo i due risultati precedenti per restituire gli ssn dei manager con familiari a carico;
- Troviamo il nome e il cognome dei managers calcolati precedentemente tramite join alla tabella *employee*.

2.5 Popolazione casuale DB

Lo scopo della lezione è quello di andare a riempire un database con dei dati casuali. I database sono principalmente di due categorie:

- MYISAM
- INNODB
- **MyISAM** non utilizza le chiavi esterne nelle sue relazioni e non esistono le transazioni ma ha il vantaggio di essere molto veloce, inoltre, ogni tipo di dato viene rappresentato attraverso 3 diversi file:
 - .frm → dove viene rappresentata la parte strutturale;
 - .myd → dove vengono salvati i dati;
 - .myi → dove vengono salvati gli indici relativi alla tabella INNODB.
- **INNODB** utilizza delle tabelle più complete ma allo stesso tempo più lente, però al contrario di MyISAM permette l'utilizzo di chiavi esterne e transazioni (commit e rollback).

Le Transazioni sono delle operazioni atomiche che vengono eseguite sul DB come per esempio un bonifico, l'acquisto di un biglietto ecc. per default tutti gli aggiornamenti sono istantanei durante le transazioni, per evitare l'aggiornamento automatico bisogna cambiare il valore di AUTOCOMMIT.

Nella creazione del DB ci troveremo ad avere delle tabelle in cui saranno presenti delle chiavi esterne, ciò che succede quando una tupla relativa ad una chiave esterna viene cancellata deve essere impostato durante la creazione del database utilizzando la clausola ON DELETE seguita da uno dei seguenti valori:

- CASCADE: viene cancellata anche la tupla della tabella in cui è presente la chiave esterna;
- SET NULL viene impostato il valore della chiave esterna a NULL;
- SET DEFAULT: viene impostato il valore della chiave esterna al valore di default impostato;
- RESTRICT/NO ACTION: si verifica prima o dopo aver tentato di aggiornare la casella.

2.5.1 Tipi di dato in MySQL

- Numerici:

- BIT[(M)];
- TINYINT[(M)] [UNSIGNED] [ZEROFILL];
- SMALLINT[(M)] [UNSIGNED] [ZEROFILL];
- MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL];
- INT[(M)] [UNSIGNED] [ZEROFILL];
- BIGINT[(M)] [UNSIGNED] [ZEROFILL];
- FLOAT[(M,D)] [UNSIGNED] [ZEROFILL];
- DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL];

- DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
- Alfanumerici:
 - [NATIONAL] CHAR(M) [BINARY — ASCII — UNICODE];
 - [NATIONAL] VARCHAR(M) [BINARY];
 - BINARY(M);
 - VARBINARY(M);
 - TINYBLOB;
 - TINYTEXT;
 - BLOB[(M)];
 - TEXT[(M)];
 - MEDIUMBLOB;
 - MEDIUMTEXT;
 - LONGBLOB;
 - LONGTEXT;
 - ENUM('valore1','valore2', ...);
 - SET('valore1','valore2', ...)
- Date e tempo:
 - DATE;
 - DATETIME;
 - TIMESTAMP[(M)];
 - TIME - YEAR[(2—4)]

2.5.2 Attributi

Oltre ad impostare il valore di ogni colonna possiamo impostare altri valori tra cui:

- Primary Key: rappresenta quale campo viene usato come chiave primaria nella tabella, deve essere necessariamente unica e usando l'attributo AUTOINCREMENT viene incrementata automaticamente ad ogni inserimento usando il primo valore libero;
- UNIQUE: può essere impostato anche se il campo non è una chiave primaria e richiede che ogni valore sia diverso dagli altri;
- INDEX: velocizza l'accesso ai dati generando degli indici, a differenza della PRIMARY KEY non è un valore unico.

2.5.3 Funzioni di Aggregazione

La clausola GROUP BY serve a specificare quali sono i campi sui cui effettuare i raggruppamenti: il motore di query, per ogni riga esaminerà tali campi e la classificherà nel gruppo corrispondente. Si possono specificare calcoli da effettuare per ogni gruppo. Esempi di operazioni sono:

- DISTINCT: visualizza solo valori distinti nel raggruppamento;
- COUNT(*): conta il numero di occorrenze nel raggruppamento;
- AVG: calcola la media dei valori;
- SUM: somma i valori del raggruppamento.

Con la clausola HAVING possiamo imporre delle condizioni ai soli raggruppamenti e ai soli campi per cui è stata utilizzata la clausola GROUP BY.

2.5.4 Popolazione del DataBase

- Creare nel DB la tabella che si vuole riempire. Nel nostro caso, usiamo una tabella di esempio, Persona, con gli attributi che si vedono nella figura.

#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(11)			No	None		Primary Spatial Distinct values
2	name	varchar(45)			No	None		Primary Spatial Distinct values
3	surname	varchar(45)			No	None		Primary Spatial Distinct values
4	region	varchar(45)			No	None		Primary Spatial Distinct values
5	province	varchar(45)			No	None		Primary Spatial Distinct values
6	city	varchar(45)			No	None		Primary Spatial Distinct values
7	cap	int(11)			No	None		Primary Spatial Distinct values
8	street	varchar(45)			No	None		Primary Spatial Distinct values
9	number	int(11)			No	None		Primary Spatial Distinct values
10	birthdate	date			No	None		Primary Spatial Distinct values

Check all With selected: Add to central columns

Figura 2.34: phpMyAdmin - Tabella di esempio

- Creare un novo foglio, in Excel, in cui andremo a generare i valori casuali da inserire nel DB.

A	B	C	D	E	F	G	H	I	J
1	ID	name	surname	region	province	city	cap	street	number
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									
20									
21									
22									
23									

Figura 2.35: Microsoft Excel - Foglio di lavoro

- Fare il download delle seguenti liste:
 - Lista nomi italiani: <https://gist.github.com/pdesterlich/2562329>;
 - Lista cognomi italiani: <https://gist.github.com/pdesterlich/2562407>;
 - Lista comuni italiani, con relativi CAP, province e regioni:
<http://lab.comuni-italiani.it/download/comuni.html> (Quest'ultimo file è in formato CSV, quindi si può aprire con Excel).
- Importare nel nostro file Excel, le liste sopraelencate:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
48															
49	conteggio	nome		conteggio	cognome		conteggio	cap	città		conteggio	provincia		conteggio	regione
50	8913	abaco		150	Agostini		8092	35031	Abano Terme		110	PD		20	VEN
51		abbondanza			Aiello			26834	Abbadia Cerreto			LO			LOM
52		abbondanzia			Albanese			23821	Abbadia Lariana			LC			TOS
53		abbondanzio			Amato			53021	Abbadia San Salvatore			SI			SAR
54		abbondazio			Antonelli			9071	Abbasanta			OR			ABR
55		abbondia			Arena			65020	Abbadigli			PE			BAS
56		abbondina			Baldi			20081	Abbiategrosso			MI			SIC
57		abbondio			Barbieri			51021	Abetone			PT			PUG
58		abdenago			Barone			85010	Abriola			PZ			PIE
59		abdon			Basile			97011	Acate			RG			LAZ
60		abdone			Battaglia			71021	Accadia			FG			CAM
61		abela			Bellini			12021	Acceglio			CN			MAR
62		abelarda			Benedetti			75011	Accettura			MT			CAL
63		abelardo			Bernardi			67020	Acciano			AQ			UMB
64		abele			Bianchi			2011	Accumoli			RI			MOL
65		abelina			Bianco			85011	Acerenza			SA			EMR
66		abelino			Brambilla			84042	Acerno			NA			FVG
67		aberardo			Bruni			80011	Acerra			CT			LIG
68		abilio			Bruno			95020	Aci Bonacorsi			MC			TAA
69		abondio			Calabrese			95021	Aci Castello			FR			VDA
70		abrama			Caputo			95022	Aci Catena			CS			
71		abramina			Carbone			95025	Aci Sant'Antonio			BS			

Figura 2.36: Microsoft Excel - Importazione liste

Nelle celle A, D, G, K, N, c'è il numero totale di elementi delle liste che si trovano nelle colonne successive a quelle indicate.

- Premere Alt + F11 per aprire la finestra che permette di implementare le nostre funzioni per generare i dati casuali.
- Cliccare con il tasto destro nella sezione indicata e selezionare inserisci modulo:

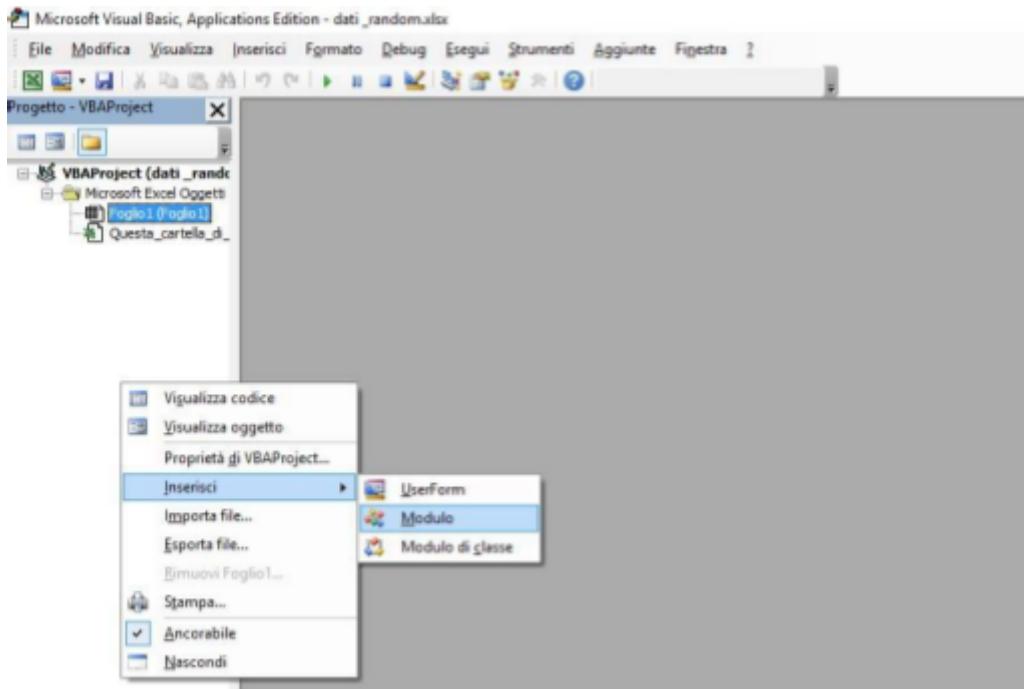


Figura 2.37: Microsoft Excel - Visual Basic for Applications

- Una volta creato il nuovo modulo, è possibile implementare le nostre funzioni per la generazione di dati casuali:

```

Microsoft Visual Basic, Applications Edition - dati_random.xls
File Modifica Visualizza Inserisci Formato Debug Esegui Strumenti Aggiunge Finestra
Progetto - VBAProject
  Microsoft Excel Oggetti
    Foglio1 (Foglio1)
    Questa_cartella_d...
  Module1

dati_random.xls - Module1 (codice)
(Generale) random_name()
Function random_name()
    a = Rnd()
    b = Range("a50").Value
    x = 50 + Int(a * b)
    r = Range("B" & CStr(x))
    random_name = r
End Function

Function random_surname()
    a = Rnd()
    b = Range("d50").Value
    x = 50 + Int(a * b)
    r = Range("E" & CStr(x))
    random_surname = r
End Function

Function random_date_between(x, y) As Date
    a = Rnd()
    b = DateDiff("d", x, y)
    r = CDate(x) + CDate(Int(a * b))
    random_date_between = r
End Function

```

Figura 2.38: Visual Basic for Applications - Nuovo Modulo

Nel nostro caso abbiamo bisogno delle seguenti funzioni:

```

1  Function random_name()
2      a = Rnd()
3      b = Range("a50").Value
4      x = 50 + Int(a * b)
5      r = Range("B" & CStr(x))
6      random_name = r
7  End Function
8
9  Function random_surname()
10     a = Rnd()
11     b = Range("d50").Value
12     x = 50 + Int(a * b)
13     r = Range("E" & CStr(x))
14     random_surname = r
15  End Function
16
17  Function random_date_between(x, y) As Date
18      a = Rnd()
19      b = DateDiff("d", x, y)
20      r = CDate(x) + CDate(Int(a * b))
21      random_date_between = r
22  End Function
23
24  Function cap(x)
25      i = 50 For Each c In [i50:i8141]
26          If c.Value = x.Value Then
27              y = i
28          i = i + 1
29      Next
30      cap = Range("h" & CStr(y)).Value
31  End Function
32
33  Function random_city()
34      a = Rnd()
35      b = Range("g50").Value
36      x = 50 + Int(a * b)
37      r = Range("i" & CStr(x))
38      random_city = r
39  End Function
40
41  Function random_region()
42      a = Rnd()
43      b = Range("n50").Value
44      x = 50 + Int(a * b)
45      r = Range("o" & CStr(x))
46      random_region = r
47  End Function
48
49  Function random_province()
50      a = Rnd()
51      b = Range("k50").Value
52      x = 50 + Int(a * b)
53      r = Range("l" & CStr(x))
54      random_province = r
55  End Function

```

Bisogna fare attenzione agli indici delle celle utilizzati, se le liste sono state inserite in una posizione differente, è necessario aggiornare gli indici nelle funzioni che sono scritte sopra.

- A questo punto è sufficiente utilizzare in Excel, le funzioni appena implementate per generare i dati.

	A	B	C	D	E	F	G	H	I	J
1	ID	name	surname	region	province	city	cap	street	number	birthdate
2	1	diaz	Milani	LAZ	AO	Taviglianc	13811	Franco	28	19/12/1969
3	2	ermo	Caruso	CAM	MN	Olginate	23854	Fontana	18	25/10/2007
4	3	gradisca	Cirillo	UMB	AR	Pantigliat	20090	Poli	12	03/09/1960
5	4	raffaeline	Conte	VDA	PT	Baragliano	85050	Pozzi	13	20/01/1975
6	5	giovangaston	Bruni	TOS	FR	Atella	85020	Parisi	3	07/03/1985
7	6	livenzo	Coppola	LAZ	AR	Salvitelle	84020	Gallo	18	09/08/2009
8	7	oscarino	Bernardi	CAL	LU	Albidona	87070	Giordano	38	13/07/1976
9	8	emerino	Vitali	FVG	CH	Isola del F	61030	Pellegrin	29	17/04/1968
10	9	fidenzio	Giorgi	TAA	AQ	Pieve d'O	26040	Ferro	38	22/11/1956
11	10	cleria	Bellini	PIE	BR	Montemo	63088	Lombardi	29	20/03/2014
12	11	deanna	Fumagalli	PUG	CB	Mileto	89852	Caputo	2	20/02/1984
13	12	venicia	Marlino	TAA	SS	Levico Ter	38056	Pace	35	04/04/1954
14	13	ottaviana	Palmieri	LAZ	CT	Casto	25070	Ferraro	35	10/10/2001
15	14	alpino	Lorusso	EMR	GR	Emarèse	11020	Villani	46	30/01/2003
16	15	oberdina	Santini	LIG	TO	Burolo	10010	Valentini	41	15/08/2002
17	16	olide	Gargiulo	VEN	MC	Capua	81043	Longo	35	19/10/1976
18	17	asia	Donati	CAL	OG	Merone	22046	Cirillo	42	05/03/2009
19	18	fiordalise	Ferraro	EMR	CR	Loano	17025	De Angeli	17	22/12/2007
20	19	melchidesec	Fumagalli	PUG	PG	Ollolai	8020	Silvestri	5	19/02/1984
21	20	fedalba	Ruggeri	BAS	AP	Casalangu	66031	Ferro	27	02/03/1950
22	21	ortesio	Romano	BAS	TV	Grottaglie	74023	Ricciardi	23	16/12/1998
23	22	gervasia	Bellini	PIE	RO	Ripalimos	86025	Ferrara	46	20/09/2002
24	23	cadira	Mazza	VDA	LE	Valle di Ci	39030	Bruni	30	17/12/2014

Figura 2.39: Microsoft Excel - Tabella popolata

- In B → $F(x) = \text{random_name}();$
- In C → $F(x) = \text{random_surname}();$
- In D → $F(x) = \text{random_region}();$
- In E → $F(x) = \text{random_province}();$
- In F → $F(x) = \text{random_city}();$
- In G → $F(x) = \text{cap}(x);$
- In H → $F(x) = \text{random_surname}();$
- In I → $F(x) = \text{CASUALE.TRA}(1; 50);$
- In J → $F(x) = \text{random_date_between}("01/01/1950"; "01/01/2016").$

Lorenzo Caputo
Mattia Marzano
27/10/2016

2.6 MySQL

- Tipi di tabelle (motori MyISAM e InnoDB);
- Tipi di dati (numerici, alfanumerici, temporali):
 - Con particolare attenzione alla gestione delle date;
- Funzioni di aggregazione:
 - MIN;
 - MAX;
 - AVG;
 - SUM;
 -
- Indici.

In un DBMS in generale e in MySQL in particolare, i dati sono organizzati in:

- Database;
- Tabelle.

2.6.1 Tipi di tabelle

I tipi di tabelle (o *storage engine*) sono dei moduli software che si occupano della memorizzazione e del recupero delle informazioni.

- **MyISAM** (motore di default dalla versione 3.23 fino alla 5.1);
- **InnoDB** (motore di default a partire dalla versione 5.0).

MyISAM

- Tabelle storiche di MySQL;
- Garantiscono un'ottima affidabilità e velocità;
- Vantaggio: la possibilità di poter utilizzare indici FULLTEXT per ricerche con ranking stile Google;
- Mancano però di alcune caratteristiche molto importanti nelle basi di dati:
 - Supporto alle chiavi esterne per garantire l'integrità referenziale;
 - Supporto alle transazioni;
- Non sono adatte per realizzare sistemi di commercio elettronico o altre applicazioni enterprise;
- Il metodo di salvataggio dei dati è basato sulla costruzione e lavorazione di 3 file binari per ogni tabella:

- nome_tabella.frm: struttura della tabella (dimensioni e tipo di ogni colonna, indici, ...);
 - nome_tabella.MYD: file che contiene tutti i dati della tabella;
 - nome_tabella.MYI: file che contiene i dati relativi agli indici del database;
- Per trasferire/copiare una o più tabelle da una macchina all'altra o da un database a un altro è sufficiente spostare questi 3 file → backup semplice;
 - Si affida al sistema operativo per il caching di letture e scritture sulle tabelle.

Le tabelle possono avere un formato:

- **Statico:** (quando la tabella non contiene colonne a lunghezza variabile es. varchar; offre maggiore sicurezza e velocità ma richiede più spazio sul disco);
- **Dinamico:** (quando la tabella contiene colonne a lunghezza variabile; può però portare a una frammentazione della tabella) → è bene effettuare periodicamente un'ottimizzazione con il comando *OPTIMIZE TABLE*;
- **Compresso:** (utile per generare tabella a sola lettura che minimizzano l'occupazione di spazio; compressione con l'utilità *myisampack* e decompressione con l'utilità *myisamchk*)

InnoDB

- Tabelle molto più complete delle MyISAM ma più lente a causa delle funzionalità aggiuntive di cui dispongono;
- Vantaggi:
 - Supporto per le chiavi esterne (*foreign key*):
 - * Le tabelle sono in grado di gestire l'integrità referenziale tra le chiavi esterne del database;
 - Supporto per la transazionalità (fondamentale per le applicazioni di commercio elettronico in cui, finché non arriva conferma da parte della banca o di chi valida la carta di credito, le query non devono essere “eseguite realmente”):
 - * Supportano commit e rollback;
 - * Sono in grado di conservare i dati dopo un eventuale crash;
 - * Supportano il lock delle colonne → aumento della produttività e dell'efficienza nel caso di utilizzo simultaneo dei database da parte di più utenti;
- Per trasferire una o più tabelle da un server all'altro non è sufficiente spostare i file → procedura di backup più complicata (si usa l'utilità *mysqldump*);
- Gestione propria della cache → modifiche dei dati più rapide (i dati modificati non vengono inviati al sistema).

```

    directory (già esistente) per il file dei dati se non indicata
    innodb_data_home_dir = /ibdata
    innodb_data_file_path=ibdata1:50M;ibdata2:50M:autoextend:max:500M
    innodb_buffer_pool_size=70M
    innodb_additional_mem_pool_size=10M
    innodb_log_group_home_dir = /iblogs
    innodb_log_files_in_group = 2
    innodb_log_file_size=20M
    innodb_log_buffer_size=8M
    innodb_flush_log_at_trx_commit=1
    innodb_lock_wait_timeout=50
    skip_external_locking
    max_connections=200
    read_buffer_size=1M
    sort_buffer_size=1M

    directory per i
    file di log (se
    non indicata si
    usa lo stesso dei
    dati)

    dovebbe essere circa
    la metà della memoria del computer

    dovebbe essere circa il 25% della
    dimensione del buffer pool

```

Figura 2.40: InnoDB - Opzioni di configurazione

Se nulla viene specificato di default MySQL crea nella directory dei dati:

- Un file dati di 10MB;
- Due file di log da 5MB.

InnoDB - Le Transazioni

- Una transazione è una sequenza di operazioni sul DB cui si vogliono associare particolari caratteristiche di correttezza, robustezza, isolamento (es. bonifico bancario, acquisto biglietto, prenotazione aerea, ...);
- Ogni connessione a MySQL inizia in autocommit mode e cioè tutte le istruzioni di aggiornamento vengono rese effettive immediatamente;
- Disattivando l'autocommit con *SET AUTOCOMMIT = 0*:
 - Le modifiche diventeranno operative solo all'esecuzione dell'istruzione *COMMIT* (conferma l'operazione);
 - Eseguendo una *ROLLBACK* invece (aborto operazione), verranno annullate tutte le modifiche fatte fino alla *COMMIT* precedente;
- Invece di disattivare l'autocommit, si possono utilizzare le transazioni iniziandole con *START TRANSACTION* o *BEGIN* e terminandole con *COMMIT* o *ROLLBACK*.

InnoDB - le chiavi esterne

- È possibile definire le foreign key, cioè collegare i valori delle colonne che contengono chiavi di altre tabelle alle tabelle stesse;
- In questo modo è possibile verificare automaticamente quando i valori della tabella madre vengono modificati o eliminati in modo da:
 - impedire queste modifiche;
 - o modificare di conseguenza anche i valori sulla tabella figlia

Customers		
ID	Last Name	First Name
001	Bond	James
024	Bauer	Jack
123	Smith	Jane

Orders			
ID	Cust_ID	Item	Qty
1	001	Fancy Gadget	4
2	024	Hand Gun	1
3	024	Bullet-proof vest	1

Figura 2.41: InnoDB - Foreign Keys

```

CREATE TABLE madre(id INT NOT NULL,
                   PRIMARY KEY (id)
) ENGINE=INNODB;

CREATE TABLE figlia(id INT, madre_id INT,
                    INDEX par_id (madre_id),
                    FOREIGN KEY (madre_id) REFERENCES madre(id)
                    ON DELETE CASCADE
) ENGINE=INNODB;

```

Figura 2.42: InnoDB - Foreign Keys - Codice

- Definizione di due tabelle (madre e figlia);
- La figlia ha una chiave esterna sulla tabella madre;
- Quando viene cancellata una riga dalla tabella madre, se il valore di id è presente in un campo madre_id della tabella figlia, la riga corrispondente verrà eliminata.

ON DELETE / ON UPDATE:

- **CASCADE**: la cancellazione o modifica di un record nella tabella madre genererà la cancellazione o la modifica dei record collegati nella tabella figlia;
- **SET NULL**: in caso di eliminazione o modifica di un record nella tabella madre i record collegati della tabella figlia verranno modificati impostando il campo NULL (opzione attivabile solo se il campo interessato della tabella figlia non è impostato a NOT NULL);
- **SET DEFAULT**: in caso di eliminazione o modifica di un record nella tabella madre i record collegati della tabella figlia verranno impostati al valore di default;
- **RESTRICT**: non permette cancellazioni sulla tabella madre se ci sono valori presenti nella tabella figlia;
- **NO ACTION**: non permette cancellazioni sulla tabella madre se ci sono valori presenti nella tabella figlia.

Le opzioni NO ACTION e RESTRICT sono tra loro molto simili: in entrambi i casi, se la verifica del vincolo fallisce, l'operazione porta ad un errore.

La principale differenza tra le due è che con l'opzione NO ACTION, la verifica del vincolo di integrità è fatta *dopo* aver tentato di modificare la tabella esterna, mentre con l'opzione RESTRICT la verifica è fatta **prima** di tentare l'esecuzione del DELETE.

2.6.2 AUTO-INCREMENT

- MyISAM gestisce una colonna di tipo auto-increment per ogni tabella:
 - incrementa automaticamente il suo valore per ogni riga scritta;
 - i valori eliminati non vengono riutilizzati nemmeno se sono gli ultimi della sequenza;
- InnoDB gestisce il valore auto-increment in modo particolare:
 - calcola il valore la prima volta che si rende necessario dopo l'avvio del server selezionando il valore massimo esistente sulla tabella e incrementandolo di 1;
 - il valore viene conservato in memoria ma non scritto sul disco per cui al riavvio successivo sarà ricalcolato → se fossero cancellati ad esempio gli ultimi valori della tabella e non venissero effettuati nuovi inserimenti, al successivo riavvio il server utilizzerà quei valori.

Quale motore utilizzare? DIPENDE!

- InnoDB: ideale per applicazioni dove il data integrity è importante e dove ci sono molteplici inserimenti e update;
- MyISAM è veloce e ideale per applicazioni dove vengono effettuate molteplici select e poco dipendenti dal data integrity.

Qual'è il motore di default?

- Comando SQL:

1 SHOW ENGINES

- File di configurazione *my.ini*:

1 default-storage-engine = MyISAM # o InnoDB

Che può essere usato anche per impostare il default storage engine (oppure default-table-type)

- Alla creazione della tabella:

1– **CREATE TABLE** tabella (a INT) ENGINE = INNODB;

- Modifica del motore:

1– **ALTER TABLE** tabella ENGINE = INNODB;

2.6.3 Tipi di dato in SQL

- Tipo numerico:

Data type	Description	
TINYINT(size)	-128 to 127 normal, 0 to 255 UNSIGNED*. The maximum number of digits may be specified in parenthesis	1 byte
SMALLINT(size)	-32768 to 32767 normal, 0 to 65535 UNSIGNED*. The maximum number of digits may be specified in parenthesis	2 byte
MEDIUMINT(size)	-8388608 to 8388607 normal, 0 to 16777215 UNSIGNED*. The maximum number of digits may be specified in parenthesis	3 byte
INT(size)	-2147483648 to 2147483647 normal, 0 to 4294967295 UNSIGNED*. The maximum number of digits may be specified in parenthesis	4 byte
BIGINT(size)	-9223372036854775808 to 9223372036854775807 normal, 0 to 18446744073709551615 UNSIGNED*. The maximum number of digits may be specified in parenthesis	8 byte
FLOAT(size,d)	A small number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter	4 byte
DOUBLE(size,d)	A large number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter	8 byte
DECIMAL(size,d)	A DOUBLE stored as a string , allowing for a fixed decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter	size + 2 byte

Figura 2.43: MySQL - Tipo di dato Numerico

- Tipo alfanumerico:

Data type	Description	
CHAR(size)	Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. Can store up to 255 characters	255 byte
VARCHAR(size)	Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. Can store up to 255 characters. Note: If you put a greater value than 255 it will be converted to a TEXT type	255 byte
TINYTEXT	Holds a string with a maximum length of 255 characters	255 byte
TEXT	Holds a string with a maximum length of 65,535 characters	65535 byte
BLOB	For BLOBs (Binary Large Objects). Holds up to 65,535 bytes of data	65535 byte
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters	1.6 MB
MEDIUMBLOB	For BLOBs (Binary Large Objects). Holds up to 16,777,215 bytes of data	1.6 MB
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters	42 MB
LONGBLOB	For BLOBs (Binary Large Objects). Holds up to 4,294,967,295 bytes of data	42 MB
ENUM(x,y,z,EE.)	Let you enter a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. Note: The values are sorted in the order you enter them. You enter the possible values in this format: ENUM('X','Y','Z')	
SET	Similar to ENUM except that SET may contain up to 64 list items and can store more than one choice	

Figura 2.44: MySQL - Tipo di dato Alfanumerico

- Tipo temporale:

Data type	Description	
DATE()	A date. Format: YYYY-MM-DD Note: The supported range is from '1000-01-01' to '9999-12-31'	3 byte
DATETIME()	*A date and time combination. Format: YYYY-MM-DD HH:MM:SS Note: The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'	8 byte
TIMESTAMP()	*A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD HH:MM:SS Note: The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC	4 byte
TIME()	A time. Format: HH:MM:SS Note: The supported range is from '-838:59:59' to '838:59:59'	3 byte
YEAR()	A year in two-digit or four-digit format. Note: Values allowed in four-digit format: 1901 to 2155. Values allowed in two-digit format: 70 to 69, representing years from 1970 to 2069	

Figura 2.45: MySQL - Tipo di dato Temporale

- Esiste pochissima coerenza nelle funzioni di data e ora tra i diversi produttori di DBMS;
- In gran parte questo dipende dal fatto che molti produttori hanno sviluppato i tipi di dati per data e ora prima dello sviluppo degli standard.

Funzioni data/ora:

- SQL Server:

FUNZIONE	SCOPO	PARAMETRI DI INPUT
DATEADD	Facilita una nuova data/ora calcolata sommando un intervallo alla parte data della data fornita	parte_data, quantità intervallo, data/ora
DATEDIFF	Facilita il numero di delimitazioni di data/ora attraversate da due date	parte_data, data/ora iniziale, data/ora finale
DATENAME	Facilita un nome di testo che rappresenta la parte_data selezionata dalla data/ora di input	parte_data, data/ora
DATENAME	Facilita un'ora che rappresenta la parte_data selezionata dalla data/ora di input	parte_data, data/ora
DAY	Facilita un'ora che rappresenta il giorno contenuto nella data/ora fornita	data/ora
GETDATE	Facilita la data/ora di sistema corrente	Nessuno
GETUTCDATE	Facilita la data/ora UTC (Universal Coordinated Time, ora coordinata universale) corrente	Nessuno
MONTH	Facilita un'ora che rappresenta il mese contenuto nella data/ora fornita	data/ora
YEAR	Facilita un'ora (quattro cifre) che rappresenta l'anno contenuto nella data/ora fornita	data/ora

Figura 2.46: SQL Server - Funzioni data/ora

- Oracle:

FUNZIONE	SCOPO	PARAMETRI DI INPUT
ADD_MONTHS	Aggiunge alla data fornita il numero di mesi forniti	data, numero di mesi (valore positivo o negativo)
CURRENT_DATE	Renditore la data corrente nel formato impostato per la sessione di database	Nessuno
EXTRACT	Lista dalla data fornita il campo di data/ora specificato	parte_data chiavi campo data/ora, data
LAST_DAY	Renditore la data fornita con il giorno specificato sull'ultimo giorno del mese	data
MONTHS_BETWEEN	Renditore il numero di mesi compresi le parti frazionarie fra due date fornite. Il risultato è negativo se la seconda data è antecedente alla prima	prima_data, seconda_data
SYSDATE	Renditore la data e l'ora di sistema corrente	Nessuno
TZ_CHAR	Guarda offuscando una data, converte la data/firma stringa di caratteri specificata da stringa_timezone	data, stringa_timezone
TZ_NESTED	Converte la stringa di caratteri fornita in una data con formattazione informa Oracle. Utilizzando stringa_timezone come modello per l'interpretazione dei confronti, stringa_timezone e stringa	data, stringa_timezone
TRUNC	Torna una data all'unità di tempo specificata nella parola chiave di campo per data/ora. Se la parola chiave viene omessa, la data viene troncata al giorno corrente	data, parola chiave campo data/ora

Figura 2.47: Oracle DB - Funzioni data/ora

- MySQL dispone di oltre 30 funzioni di data e ora. Le più utilizzate:

FUNZIONE	SCOPO	PARAMETRI DI INPUT
ADDATE	Genera due espressioni di data, intervallo o data/ora, generando una nuova data	espressione 1, espressione 2
ADTIME	Somma due espressioni di ora, generando una nuova ora	espressione 1, espressione 2
CURDATE	Riporta la data corrente nel formato AAAA-MM-GG	Nessuna
CURTIME	Riporta la parte di data di un'espressione di data/ora	espressione di data/ora
DATEDIFF	Riporta il numero di giorni tra due date	data iniziale, data finale
DATE_FORMAT	Formatta una data secondo una stringa di formato	data, stringa di formato
DAYNAME	Riporta il nome per il giorno della settimana contenuto in una data	data
DAYOFMONTH	Riporta il giorno del mese, nell'intervallo fra 1 e 31	data
DAYOFWEEK	Riporta un indice numerico per il giorno della settimana contenuto in una data (1 per domenica, 2 per lunedì e così via)	data
DAYOFYEAR	Riporta il giorno dell'anno per il giorno contenuto in una data con un intervallo valido fra 1 e 365	data
LAST_DAY	Modifica il giorno in una data nell'ultimo giorno del mese	data
MONTH	Riporta il mese contenuto in una data con un intervallo valido fra 1 e 12	data
MONTHNAME	Riporta il nome del mese contenuto in una data	data
NOW	Riporta la data e l'ora corrente	Nessuna
STR_TO_DATE	Converte una stringa di caratteri in un elemento di data in formato database; la stringa di formato indica il formato delle informazioni di data nella stringa di caratteri di input	stringa di caratteri, stringa di formato
TIME	Riporta la parte oraria di un'espressione di data/ora	espressione di data/ora
TIME_DIFOFF	Riporta la differenza oraria tra due parametri data/ora o espressi con ore	espressione 1, espressione 2
TIME_FORMAT	Formatta una data secondo la stringa di formato	ora, stringa di formato
UTC_DATE	Riporta le date UTC (Universal Coordinated Time, ora corrente universale) corrette	Nessuna
UTC_TIME	Riporta le date UTC (Universal Coordinated Time, ora corrente universale) corrette	Nessuna
WEEKOFYEAR	Riporta le settimane dall'anno, nell'intervallo fra 1 e 54	data

Figura 2.48: MySQL - Funzioni data/ora

- Le seguenti funzioni mostrano a video la data e l'ora attuale:

- 1 **SELECT CURDATE()**
- 2 **SELECT CURTIME()**
- 3 **SELECT NOW()**

che ritornano rispettivamente:

- 2013-10-02 (aaaa/mm/gg);
- 13:02:57 (hh:mm:ss);
- 2013-10-02 13:02:57 (aaaa/mm/gg hh:mm:ss).

Formattazione delle date:

Specificatore	Descrizione
%d	giorno del mese numerico 00...31
%M	nome del mese January...December
%u	mese numerico 00...12
%H	ora 00...23
%i	minuti 00...59
%s	secondi 00...59
%Y	anno di quattro cifre
%y	anno di due cifre

Figura 2.49: MySQL - Formato date

```

mysql> select DATE_FORMAT('2005-04-12 15:23:04','%d-%m-%Y %H:%i:%s') as 'data
formattata';
+-----+
| data formattata |
+-----+
| 12-04-2005 15:23:04 |
+-----+

mysql> select DATE_FORMAT('2005-04-12 15:23:04','%d %M %y, ore: %H') as 'data
formattata';
+-----+
| data formattata |
+-----+
| 12 April 05, ore: 15 |
+-----+

mysql> select
DATE_FORMAT('20050412152304','giorno:%d,mese:%m,anno:%Y,ore:%H e %i') as
'data formattata';
+-----+
| data formattata |
+-----+
| giorno:12,mese:04,anno:2005,ore:15 e 23 |
+-----+

```

Figura 2.50: MySQL - Formato date

Per la formattazione del tempo si può usare la funzione **TIME_FORMAT()**, del tutto analoga a DATE_FORMAT, ma che utilizza solo gli specificatori di formato che manipolano ore, minuti e secondi.

- Dalla versione 4.1.1 di MySQL è disponibile la funzione **STR_TO_DATE()**, inversa di DATE_FORMAT;
- Essa riceve come argomenti la stringa contenente la data e la corrispondente stringa di formattazione e restituisce un valore DATETIME, DATE o TIME a seconda delle circostanze.

```

mysql> select STR_TO_DATE('03/10/2005','%d/%m/%Y');
+-----+
| STR_TO_DATE('03/10/2005', '%d/%m/%Y') |
+-----+
| 2005-10-03 |
+-----+

mysql> select STR_TO_DATE('03.10.2005 12.33','%d.%m.%Y %H.%i');
+-----+
| STR_TO_DATE('03.10.2005 12.33', '%d.%m.%Y %H.%i') |
+-----+
| 2005-10-03 12:33:00 |
+-----+

```

Figura 2.51: MySQL - Funzione STR_TO_DATE()

Timestamp:

- Campo utile per determinare automaticamente l'istante in cui un certo record è stato inserito o modificato;
- **Nota:** se la tabella contiene più campi TIMESTAMP solo il primo sarà automaticamente aggiornato (dalla versione 4.1.2 di MySQL si può specificare quale colonna aggiornare).
- Per ottenere tale automatismo sarà sufficiente non specificare nelle istruzioni di INSERT o UPDATE il valore per la colonna in oggetto oppure impostarla a NULL;

- Il formato di visualizzazione del TIMESTAMP varia a seconda del valore specificato per “dimensione”;
- Es. considerando la data del 20 maggio 2005 19:11:02.

definizione	visualizzazione	esempio
TIMESTAMP(14)	AAAAMMDDOOGGMMSS	20050520191102
TIMESTAMP(12)	AAWMDDOOGGMMSS	050520191102
TIMESTAMP(10)	AAWMDDOOGGMM	0505201911
TIMESTAMP(8)	AAAAMMDD	20050520
TIMESTAMP(6)	AAWMDD	050520
TIMESTAMP(4)	AAWM	0505
TIMESTAMP(2)	AA	05
TIMESTAMP	AAAAMMDDOOGGMMSS	20050520191102

Figura 2.52: MySQL - TIMESTAMP()

Funzioni per la somma e la sottrazione del tempo:

Funzione	Utilizzo
DATE_ADD()	DATE_ADD(data, INTERVAL espressione tipo)
DATE_SUB()	DATE_SUB(data, INTERVAL espressione tipo)
PERIOD_ADD()	PERIOD_ADD(Periodo, Mesi) Il Periodo va espresso informato AAAANM oppure AAMM
PERIOD_SUB()	PERIOD_SUB(Periodo1, Periodo2) Il Periodo va espresso informato AAAANM oppure AAMM

Figura 2.53: MySQL - Somma e Sottrazione temporale

- Sommare 10 giorni alla data corrente:

1 **SELECT DATE_ADD(CURDATE() ,INTERVAL 10 DAYS) ;**

- Sottrarre 2 mesi da una data specifica:

1 **SELECT DATESUB('2005-01-23' ,INTERVAL 2 MONTHS) ;**

- Aggiungere 7 mesi al mese di gennaio 2005:

1 **SELECT PERIOD_ADD(200501 ,7) ;**

- Sottrarre 3 mesi al mese di gennaio 2005:

1 **SELECT PERIOD_ADD(200501 , -3) ;**

- Trovare la differenza (espressa in mesi) tra il gennaio 2005 ed ottobre 2002:

1 **SELECT PERIOD_SUB(200501 ,200210) ;**

2.6.4 Funzioni di aggregazione

- Le funzioni di aggregazione non lavorano su un solo dato ma su insiemi di dati → restituiscono un unico valore come “sintesi” di n valori;
- Le funzioni principali:
 - COUNT restituisce un intero che indica il numero di record trovati;
 - AVG restituisce la media;
 - MIN e MAX restituiscono il minore e il maggiore rispettivamente;
 - SUM restituisce la somma tra più record dello stesso campo.

```

SELECT COUNT(*) FROM tabl
-- Conta il numero di righe della tabella
SELECT COUNT(columna) FROM tabl
-- Conta i valori non NULL di colonna
SELECT COUNT(DISTINCT columna) FROM tabl
-- Conta i diversi valori (duplicati volgono per 1)
SELECT AVG(columna) FROM tabl
-- calcola la media dei valori
SELECT AVG(DISTINCT columna) FROM tabl
-- calcola la media conteggiano una sola volta i duplicati
SELECT MAX(columna) FROM tabl
-- trova il valore massimo
SELECT SUM(columna) FROM tabl
-- calcola la somma
  
```

Figura 2.54: MySQL - Fuzioni di aggregazione

- Nell'utilizzo di una funzione di aggregazione è importante (consigliato, anche se non obbligatorio) specificare un alias per il risultato, con l'utilizzo della clausola **AS**.
- Le funzioni di aggregazione vengono usate spesso in combinazione con la clausola **GROUP BY** → funzioni svolte non su tutte le righe estratte dalla WHERE, ma singolarmente su ogni gruppo di righe formato dalla GROUP BY;
- Lo standard SQL vuole che in una SELECT che contiene funzioni di aggregazione tutte le colonne su cui tali funzioni non lavorano siano comprese nella GROUP BY;
- Es. questa query non sarebbe valida: manca la GROUP BY sul campo nome:

```

SELECT ordini.idCliente, clienti.nome, MAX(pagamenti)
FROM ordini,clienti
WHERE ordini.idCliente = clienti.id
GROUP BY ordini.idCliente
  
```

Figura 2.55: MySQL - Query invalida per via della mancanza del campo nome in GROUP BY

- La query ha tuttavia un senso logico, perché il valore di nome, dipendendo dalla join, sarà sempre lo stesso in ogni gruppo di righe con lo stesso ‘idCliente’: di conseguenza è superfluo inserirlo nella GROUP BY;
- MySQL quindi ci consente di usare questa sintassi: bisogna però fare attenzione a non mettere la GROUP BY su un campo che non ha un valore unico, in quanto ne otterremmo un risultato non prevedibile.

- *COUNT*:

- Viene utilizzata per recuperare il numero di righe di una colonna;
- Può essere utilizzata su qualunque tipo di dato.

```
mysql> SELECT COUNT(*) FROM impiegati;
+-----+
| COUNT(*) |
+-----+
|      7 |
+-----+
1 row in set (0.00 sec)
```

Figura 2.56: MySQL - Funzione COUNT

Variante: COUNT(DISTINCT);

- Restituisce il numero delle diverse combinazioni che non contengono il valore NULL;
- Es. in una colonna abbiamo 10 righe: 5 contenenti la parola "calcio", 3 contenenti "tennis" e le ultime 2 con "golf". Effettuando un COUNT(DISTINCT) avremo il numero di combinazioni diverse, ovvero 3 (calcio, tennis, golf).

1 **SELECT COUNT(DISTINCT nomeCampo) FROM nomeTabella ;**

- *MAX/MIN*:

- **MAX**: restituisce il valore più alto contenuto all'interno di una colonna;
- Per i campi numerici, restituisce il numero più alto, per quelli testuali (nei nuovi MySQL questa operazione è permessa) seleziona il campo che secondo l'ordine alfabetico è più avanti:

1 **SELECT MAX(nomeCampo) FROM nomeTabella ;**

- **MIN**: fa esattamente l'opposto della precedente: prende il valore più basso:

1 **SELECT MIN(nomeCampo) FROM nomeTabella ;**

```
mysql> SELECT MIN(stipendio),MAX(stipendio) FROM impiegati;
+-----+-----+
| MIN(stipendio) | MAX(stipendio) |
+-----+-----+
|      750.25 |      2450.00 |
+-----+-----+
1 row in set (0.00 sec)
```

Figura 2.57: MySQL - MIN/MAX

- *AVG/SUM*:

- **AVG**: restituisce una media dei valori presenti in un campo (va applicata ai soli campi numerici):

```

mysql> SELECT cognome, AVG(stipendio) FROM impiegati GROUP BY cognome;
+-----+-----+
| cognome | AVG(stipendio) |
+-----+-----+
| Bianchi | 820.000000 |
| Gialli | 1250.000000 |
| Neri | 980.000000 |
| Rossi | 1600.125000 |
| Verdi | 1024.000000 |
+-----+
5 rows in set (0.00 sec)

```

Figura 2.58: MySQL - Funzione AVG

- **SUM**: somma i valori contenuti nel campo (va applicata ai soli campi numerici);

```

mysql> SELECT SUM(stipendio), AVG(stipendio) FROM impiegati;
+-----+-----+
| SUM(stipendio) | AVG(stipendio) |
+-----+-----+
| 7274.25 | 1212.375000 |
+-----+
1 row in set (0.00 sec)

```

Figura 2.59: MySQL - Funzione SUM

- *GROUP BY*:

```

mysql> SELECT qualifica,COUNT(*) AS impiegati
    -> FROM impiegati
    -> GROUP BY qualifica;
+-----+-----+
| qualifica | impiegati |
+-----+-----+
| Centralinista | 1 |
| Presidente | 1 |
| Programmatore | 2 |
| Ragioniere | 1 |
| Segretario | 2 |
+-----+
5 rows in set (0.00 sec)

mysql> SELECT ufficio,COUNT(*) AS impiegati
    -> FROM impiegati
    -> GROUP BY ufficio;
+-----+-----+
| ufficio | impiegati |
+-----+-----+
| Acquisti | 1 |
| Presidenza | 3 |
| Reception | 1 |
| Sviluppo | 2 |
+-----+
4 rows in set (0.00 sec)

mysql> SELECT ufficio,qualifica,COUNT(*) AS impiegati
    -> FROM impiegati
    -> GROUP BY ufficio,qualifica;
+-----+-----+-----+
| ufficio | qualifica | impiegati |
+-----+-----+-----+
| Acquisti | Ragioniere | 1 |
| Presidenza | Presidente | 1 |
| Presidenza | Segretario | 2 |
| Reception | Centralinista | 1 |
| Sviluppo | Programmatore | 2 |
+-----+
5 rows in set (0.00 sec)

```

Quante volte una qualifica è
presente in ogni ufficio?

Figura 2.60: MySQL - Clausola GROUP BY

```

mysql> SELECT qualifica,MIN(stipendio),MAX(stipendio),AVG(stipendio)
-> FROM impiegati
-> GROUP BY qualifica;
+-----+-----+-----+-----+
| qualifica | MIN(stipendio) | MAX(stipendio) | AVG(stipendio) |
+-----+-----+-----+-----+
| Centralinista | 900.00 | 900.00 | 900.000000 |
| Presidente | 2450.00 | 2450.00 | 2450.000000 |
| Programmatore | 1024.00 | 1250.00 | 1137.000000 |
| Ragoniere | 700.00 | 700.00 | 700.000000 |
| Segretario | 750.25 | 820.00 | 785.125000 |
+-----+-----+-----+-----+
5 rows in set (0.01 sec)

mysql> SELECT ufficio,MIN(stipendio),MAX(stipendio),AVG(stipendio)
-> FROM impiegati
-> GROUP BY ufficio;
+-----+-----+-----+-----+
| ufficio | MIN(stipendio) | MAX(stipendio) | AVG(stipendio) |
+-----+-----+-----+-----+
| Acquisti | 700.00 | 700.00 | 700.000000 |
| Presidenza | 750.25 | 2450.00 | 1340.083333 |
| Reception | 980.00 | 980.00 | 980.000000 |
| Sviluppo | 1024.00 | 1250.00 | 1137.000000 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

Figura 2.61: MySQL - Clausola GROUP BY 2

- *HAVING:*

- Usando la clausola GROUP BY, generalmente non si è interessati a tutti i risultati della selezione, ma solo ad alcuni gruppi che possiedono determinati requisiti:

```

mysql> SELECT qualifica,stipendio
-> FROM impiegati
-> WHERE ufficio = 'Presidenza';
+-----+-----+
| qualifica | stipendio |
+-----+-----+
| Presidente | 2450.00 |
| Segretario | 820.00 |
| Segretario | 750.25 |
+-----+-----+
3 rows in set (0.00 sec)

```

Figura 2.62: MySQL - Clausola WHERE

- WHERE identifica un set iniziale di record che poi vengono selezionati dalla tabella:

```

mysql> SELECT qualifica,COUNT(*) AS impiegati,AVG(stipendio) AS media_stip
-> FROM impiegati
-> WHERE ufficio = 'Presidenza'
-> GROUP BY qualifica;
+-----+-----+-----+
| qualifica | impiegati | media_stip |
+-----+-----+-----+
| Presidente | 1 | 2450.000000 |
| Segretario | 2 | 785.125000 |
+-----+-----+
2 rows in set (0.00 sec)

```

Figura 2.63: MySQL - Clausola WHERE + GROUP BY

- Aggiungendo alla query la clausola GROUP BY e qualche funzione di aggregazione, i record vengono raggruppati e per ogni gruppo vengono eseguiti i calcoli richiesti;

- Aggiungendo una clausola HAVING, viene ristretto il numero di record mostrati:

```
mysql> SELECT qualifica,COUNT(*) AS impiegati,AVG(stipendio) AS media_stip
-> FROM impiegati
-> WHERE ufficio = 'Presidenza'
-> GROUP BY qualifica
-> HAVING COUNT(*) > 1;
+-----+-----+-----+
| qualifica | impiegati | media_stip |
+-----+-----+-----+
| Segretario |        2 | 785.125000 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Figura 2.64: MySQL - Clausola HAVING

- WHERE: riduce il numero di record da processare fin dall'inizio e quindi rende la query più efficiente;
- Scegliere i valori da mostrare usando HAVING senza prima aver limitato i record da processare con WHERE, può rendere la query più pesante, perché verrebbero eseguiti raggruppamenti e calcoli effettuati dalle funzioni di aggregazione su record che non interessano;

2.6.5 phpMyAdmin – indici

- In MySQL è possibile creare diversi tipi di indice:



Figura 2.65: MySQL - Tipi di indici

- **Primary Key**: indice associato alla chiave primaria, non può contenere valori NULL né valori duplicati. Usato spesso in associazione con la direttiva auto_increment in caso di valori numerici unici;
- **Index** utilizzato per velocizzare l'accesso ai dati. A differenza della chiave primaria l'indice non è unico e può essere creato per accelerare l'elaborazione delle query. Ci possono essere valori duplicati e valori NULL;
- **Unique** come index, solo che tutti i valori del campo indicizzato univoco devono comparire solo una volta (non ci possono essere valori duplicati), sono consentiti valori NULL;
- **Fulltext** indice specializzato per la ricerca di parole nei testi. È supportato per i campi di tipo VARCHAR e TEXT.

2.6.6 Popolare database

- Supponiamo di avere un'entità Person e di volerla popolare con tanti dati random ma validi in base al tipo di attributo considerato:

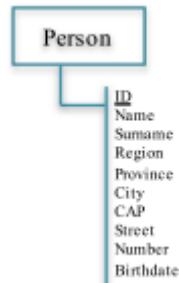


Figura 2.66: Entità Person

- Sul web esistono diverse sorgenti dati utili ai fini del popolamento di database per poter eseguire querye fare test;
- Supponiamo di voler utilizzare Excel “come appoggio”

Considerando la nostra entità ci occorre:

- Elenco di nomi <https://gist.github.com/pdesterlich/2562329>;
- Elenco di cognomi <https://gist.github.com/pdesterlich/2562407>;
- Elenco delle regioni italiane (piccolo elenco: copia e incolla dal web; ma anche con sql <https://gist.github.com/TheHiddenHaku/7103443>);
- Elenco delle province italiane
<https://gist.github.com/iamlucamilan/0727b08c2ab9a4e83cc3>;
- Elenco province e regioni
<http://www.mandile.it/wp-content/uploads/provinciaregione-sigla.csv>;
- Elenco città italiane con CAP <http://lab.comuni-italiani.it/download/comuni.html>;
- Per gli altri attributi (es. civico) usiamo funzioni di Excel = *CASUALE.TRA(1;50)*

Per la generazione di dati random è necessario scrivere qualche funzione VBA (**Visual Basic Editor**):

- Windows: Alt + F11;
- MAC Os: Fn + Alt + F11 oppure menu Strumenti → Macro → Visual Basic Editor.

```

1 Function random_name()
2     a = Rnd()
3     b = range("a50").Value
4     x = 50 + Int(a * b)
5     r = range("B" & CStr(x))
6     random_name = r
7 End Function

```

La funzione **Rnd()** restituisce un numero reale fra 0 e 1 (estremi esclusi) In generale, come trasformare il numero casuale in un numero intero compreso fra min e max? **Int(Rnd()*(max-min+1))+min.**

Per generare date di nascita random:

```

1 Function random_date_between(x, y) As Date
2     a = Rnd()
3     b = DateDiff("d", x, y)
4     r = CDate(x) + CDate(Int(a * b))
5     random_date_between = r
6 End Function

```

ID	Name	Surname	Region	Province	City	CAP	Street	Number	Birthdate
1	Modesto	Poli	Calabria	Ancona	Tiglano	73030	Rizzo	10	24/04/92
2	Ravio	Locatelli	Molise	Brindisi	Salice Salentino	73015	Ferraro	25	28/04/68
3	Giovane	Perrone	Ville d'Aosta	Piacenza	Taviano	73067	Piras	41	24/03/38
4	Bartolomeo	Messina	Lombardia	Lofina	Gagliano	73044	Bonelli	1	07/06/78
5	Domenico	Romano	Sardegna	Menza e della Br	Marsano	73025	Piazza	45	30/09/78
6	Amaranto	Barbieri	Molise	Latina	Santa Cesarea Terme	73020	Rizzi	1	24/05/96
7	Ismaila	D'Angelo	Molise	Ravenna	Taviano	73057	Palmieri	15	23/07/64
8	Annalisa	Blanco	Marche	Foggia	Minervino di Lecce	73027	Bellini	22	23/02/77
9	Teseo	Caputo	Liguria	Pesaro e Urbino	Gagliano	73013	Brusil	22	13/01/74
10	Amato	Flore	Lombardia	Caserta	San Cesario di Lecce	73012	Ruggeri	36	19/09/82
11	Fernando	Caruso	Friuli-Venezia Giulia	Campobasso	Scuinzano	73018	Pozzi	46	03/07/70
12	Corrado	Amato	Basilicata	Padova	Liguria	73069	Gallo	13	08/01/98
13	Sandro	Marina	Emilia-Romagna	Bari	Melignano	73020	Valentini	37	15/09/55
14	Oriente	Napolitane	Veneto	Lucera	Guagnano	73012	Ruggiero	8	04/03/96
15	Corrado	Martirelli	Basilicata	Treviso	Aietre	73012	Bernardi	50	04/01/82
16	Martin	Romano	Liguria	Cremena	San Cesario di Lecce	73018	Testa	22	22/05/65
17	Cesare	Brambilla	Liguria	Medio Campidano	San Cesario di Lecce	73016	Galli	33	18/06/55
18	Antabile	Coccarolo	Marche	Imperia	Alessano	73003	Marinelli	45	31/12/72
19	Adione	Olivieri	Campagna	Enna	Santerica	73030	Rossa	6	29/12/84
20	Dionigi	Levite	Marche	Udine	Nerito	73012	Benedetti	47	08/07/76
21	Pellegrino	Fusco	Friuli-Venezia Giulia	Pesaro e Urbino	Tiglano	73030	Messina	16	09/07/78
22	Durillo	Cattaneo	Puglia	Pesaro e Urbino	Spezzola	73040	Lombardi	26	12/06/75
23	Castigero	Pagan	Molise	Lecce	Andriano	73032	Ferrari	40	08/04/65
24	Giovanni	Silvestri	Campagna	Belluno	Nevoidi	73005	Montanari	41	30/05/72
25	Celestino	Malini	Ville d'Aosta	Nuvoro	Taviano	73057	More	26	28/02/76
26	Eberardo	Bruni	Sardegna	Prato	Carmignano	73041	Donati	49	29/11/71
27	Aladino	Piazza	Molise	Trieste	Montesano Salentino	73030	Farma	47	28/04/71
28	Gaetano	Giorgi	Liguria	Sandrio	Cutrofiano	73020	Romanò	26	12/05/92

Figura 2.67: Microsoft Excel - Popolamento completato

Una volta ottenuta la sorgente dati, si può salvare il file in formato csv per poi importarlo sul DB.

Lucia Vaira
27/10/2016

2.7 SQL Recap

SQL non è un vero linguaggio di programmazione perché ogni cosa può essere rappresentata come una singola espressione. Anche operazioni molto complesse possono essere ricondotte essenzialmente a un'unica espressione. Questo linguaggio si fonda quindi su un paradigma dichiarativo, come avviene in linguaggi come HTML, in cui abbiamo vari tag che specificano ciò che deve essere visualizzato sullo schermo. In SQL dobbiamo dire ciò di cui abbiamo bisogno, non come fare per ottenerlo. Eventuali ottimizzazioni delle query e le decisioni su come eseguirle

sono prese in carico dal DBMS. Abbiamo solo insiemi e operatori che ci permettono di combinare tali insiemi. **Regola Fondamentale: Non scrivete algoritmi basati su SQL.** La gestione dei dati si fa in SQL. Non usate Java per il data layer. Java, piuttosto che C++ o PHP, va bene per il presentation layer, e in parte per la business logic, ma i dati deve gestirli il DBMS. Tranne casi rarissimi, non si scrivono algoritmi di manipolazione dati per la gestione dei database.

SQL sta per Structured Query Language. Ultimo standard è SQL99. Se vi sembra obsoleto, sappiate che dentro ci sono cose così avanzate che ancora devono essere implementate tutte. La base però sta in ciò che abbiamo detto per l'algebra relazionale. Per comprendere meglio cosa è SQL, dobbiamo avere chiaro in mente il ciclo di vita dei database e il ciclo di vita dei dati. Un database è una collezione di dati e vincoli. Quando progettiamo un database lo traduciamo in tabelle e vincoli e poi lo spostiamo su una macchina fisica. Le operazioni che si possono fare sono:

Schema creation: il primo comando che impariamo è CREATE SCHEMA. Nello schema possiamo creare tabelle, con dei vincoli, delle viste (tabelle virtuali ottenute da tabelle reali attraverso operatori, come quando facciamo una join). Possiamo creare funzioni, modificarle ecc... Infine abbiamo *stored procedures* (create in un linguaggio presente nel DBMS). Sono programmi in grado di estrarre informazioni dalle tabelle, scritti in un linguaggio specifico. Tali funzioni vivono nel DBMS. Possiamo anche utilizzare linguaggi esterni tramite appositi plugin. È importante dire che oltre ai database che creiamo noi per memorizzare informazioni, esiste anche un meta-database, o **catalogo**, presente nel DBMS, che si può pensare come la collezione di tutti gli schemi che abbiamo creato, con i relativi oggetti, tabelle, funzioni, viste, ecc. Mysql è basato su un catalogo di sistema, che è un database, contenente informazioni sugli altri database. Ci sarà il tipo di entità 'entità', un altro 'vincoli di chiave esterna', ecc... Quando vogliamo informazioni sul database, interroghiamo il system catalog.

CREATE SCHEMA crea una cartella vuota in cui mettiamo tutte le nostre tabelle. Ogni schema corrisponde a un database separato. Non andiamo a mischiare informazioni non appartenenti allo specifico database. Normalmente le tabelle usano la notazione puntata. Ad esempio se vogliamo la tabella 'persona' dallo schema 'amici', scriviamo SELECT from amici.persona.

I DBMS non solo hanno tabelle, ma sono in grado di gestire utenti e gruppi, ciascuno col suo set di autorizzazioni. Se ad esempio scriviamo:

1 CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith'

allora Jsmith sarà il proprietario dello schema creato e avrà privilegi esclusivi di accesso e modifica. Se non si specifica l'autorizzazione, tipicamente l'owner del database è autorizzato a far tutto. Quando lo schema è pronto, possiamo creare e modificare le tabelle. CREATE TABLE COMPANY.EMPLOYEE crea la tabella EMPLOYEE nel database COMPANY. Quando le tabelle sono create, allora operiamo con un set differente di comandi. Inizia il ciclo di vita dei dati. Abbiamo a disposizione differenti tipi di dati:

- Numerici: SMALLINT corrisponde a 1 o 2 byte, int 2 o 4 byte, LONGINT fino a 8 byte. Quando facciamo funzioni di aggregazione possiamo ottenere overflow se non dimensioniamo bene i tipi di dato. Si specifica anche la cifra di precisione. Vi sono poi i numeri in virgola mobile, tipo FLOAT o REAL o DOUBLE. Nel passaggio di dati da una macchina all'altra bisogna stare attenti alla compatibilità delle due macchine;
- Stringhe: I gigabyte costano niente per cui è abbastanza facile avere memoria. Un singolo byte può fare la differenza quando è moltiplicato per milioni di volte. VARCHAR (1000) vuol dire che se mettiamo un nome di 10 lettere, non consumiamo 1000, perché VARCHAR si adatta alle dimensioni. Ma è anche time-consuming. Se il database deve essere veloce usiamo CHARACTER. Con CLOB ogni cella può memorizzare miliardi di caratteri. È utile per grandi documenti tipo pdf;

- Bit string: usate essenzialmente per dati provenienti da apparecchiature.
- Boolean: TRUE o FALSE;
- DATE: la data ha dieci posizioni distinte. Timestamp contiene la data e ora completa e consente anche di specificare la time zone.

Esiste anche un meccanismo di estensione CREATE DOMAIN, che è solo un modo diverso di assegnare valori predefiniti a certi tipi di dato, come si fa con typedef in C.

2.7.1 VINCOLI

- **Default value:** valore che assume il dato se non viene specificato;
- Un vincolo interessante è il **CHECK**: quando inseriamo un dato viene sempre verificato che sia rispettato quel check. Ad esempio, tutte le volte che si inserisce un nuovo dipendente, verifica che abbia più di 18 anni, altrimenti non lo assumi. Tale controllo può essere esteso tramite i trigger, che agiscono sull'intera tabella, ad esempio si può dire ‘non inserire questo dipendente se l'età media supera un certo valore’;
- **Integrità referenziale:**

Prima di tutto, posso dire che un certo attributo è unico; il concetto di unicità è associato alla chiave primaria. Si può anche introdurre una chiave secondaria che sia unica. Per quanto riguarda le chiavi esterne, si possono specificare vari tipi di azioni: ON UPDATE CASCADE, SET DEFAULT, SET NULL. Se cancello una merce, cancella tutte le vendite. Questo è il **cascade**. Potrei anche dire on delete set null, oppure set default, ovvero tutte le volte che cancello un cliente, assegna le vendite a un cliente generico. Si tratta di operazioni fatte in automatico quando si cancella o modifica un dato con vincoli di integrità referenziale.

I vincoli possono essere messi alla fine di ogni tabella. Oppure posso dargli un nome in modo che posso richiamarlo in un ALTER CONSTRAINT per modificarlo in seguito. È buona norma dare un nome per evitare di riscrivere tutto.

2.7.2 Meccanismo base Query

Il meccanismo di base delle query è basato sul costrutto:

- 1 **SELECT** <attribute list>
- 2 **FROM** <table list>
- 3 **WHERE** <condition>

Ad esempio,

- 1 **SELECT** Fname, Lname, Address
- 2 **FROM** EMPLOYEE, DEPARTMENT
- 3 **WHERE** Dname = 'Research' **AND** Dnumber = Dno;

Restituisce nome e indirizzo di tutti gli impiegati che lavorano nel dipartimento “Ricerca”. Si dice che Dname = ‘Research’ è una selection condition, mentre Dnumber = Dno è una join condition. Per evitare ambiguità con gli attributi, si può preporre un attributo con il nome della relazione, avvalendosi della notazione puntata. Per abbreviare, si può rinominare la relazione con un **alias**.

Infine, alle tabelle si possono applicare le normali funzioni insiemistiche come UNION o INTERSECT, con la possibilità di specificare se si vogliono elementi tutti distinti, o si permettono i duplicati.

2.7.3 ESERCIZIO

Si vuole progettare una base di dati per la raccolta e la gestione di informazioni relative al contesto di una specifica stagione di Formula 1. A tale scopo sono stati ricavati i seguenti requisiti:

- Calendario Competizione;
- Piloti;
- Veicoli;
- Classifica Piloti;
- Classifica Marche;
- Circuiti → Posti disponibili → Clienti;
- Meteo;
- Team

Essendo il primo requisito equivoco sotto certi aspetti (accorpamento dei propri attributi in altri requisiti), si è passati alla trattazione del secondo, identificandolo come un'entità e ricavandone i relativi attributi.

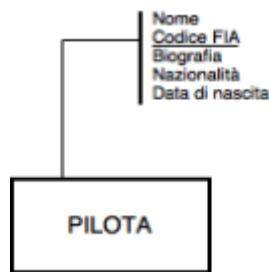


Figura 2.68: Entità PILOTA

In seguito si è associata l'entità *PILOTA* all'entità *VEICOLO* (derivante dal requisito 3) tramite la relazione *GUIDA*, contenente informazioni su data, ora e setup del veicolo, che sarà soggetto a variazioni durante lo svolgimento delle varie corse.

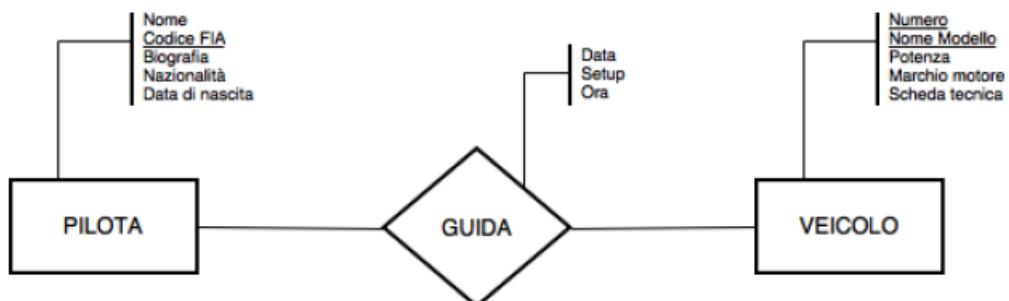


Figura 2.69: Relazione Pilota-Guida-Veicolo

Dalla discussione dei requisiti successivi è emerso che:

- I requisiti Team, Corsa e Circuito sono stati identificati come entità;
- Il requisito meteo è stato considerato attributo dell'entità *CORSA*;
- allo stesso modo il requisito Posti disponibili è stato trattato come attributo composto (Categoria + Numero) dell'entità *CIRCUITO*;
- I requisiti sulla Classifica sono stati ritenuti calcolabili a partire dalle informazioni già contenute nel database;
- Tra *PILOTA* e *CORSA* una doppia relazione (*PARTECIPA* e *SI QUALIFICA*), derivante dall'implementazione del pattern Preventivo-Consuntivo.

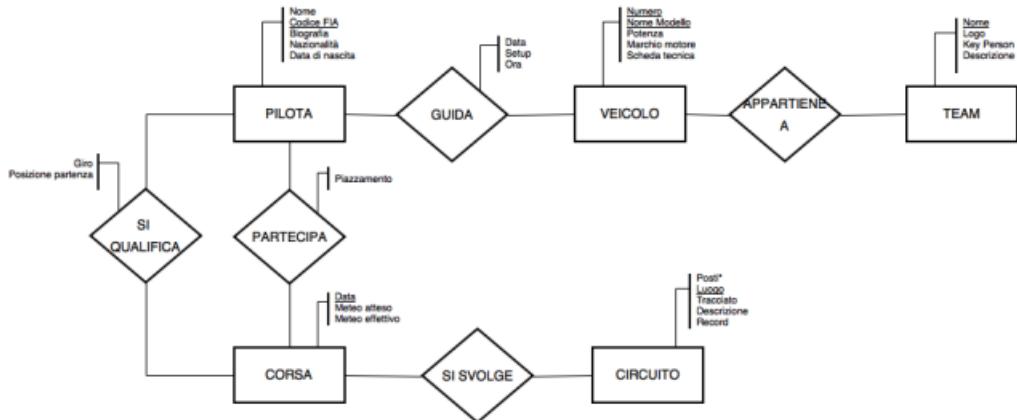


Figura 2.70: Formula 1 - Incremental ER

Ulteriori osservazioni hanno portato alla creazione di un'entità *EVENTO* che tiene conto di accadimenti talvolta sporadici che possono coinvolgere piloti e veicoli in una corsa.

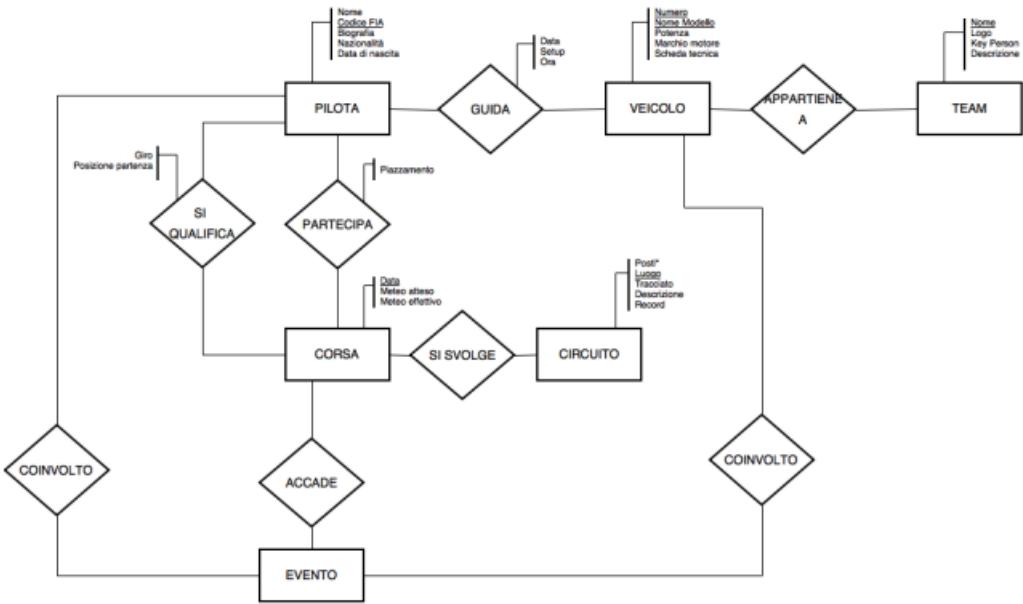


Figura 2.71: Formula 1 - Ultime modifiche

Infine il requisito Cliente è stato identificato come entità e associato a *CORSA* tramite il pattern Preventivo-Consuntivo. In conclusione si noti che la relazione tra *CORSA* e *CIRCUITO* è unaria in quanto ogni singola corsa si svolge su un singolo circuito, pervenendo alla seguente soluzione finale:

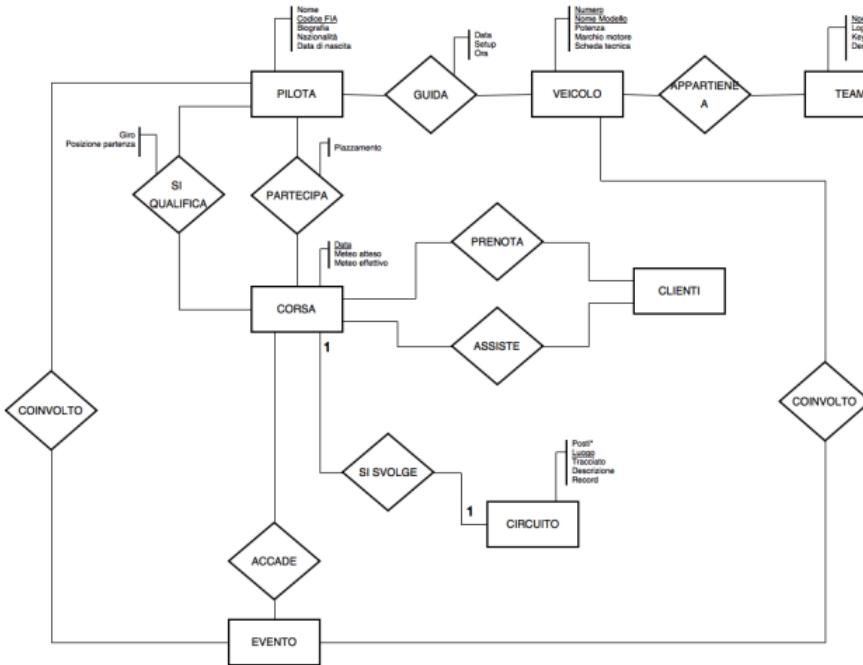


Figura 2.72: Formula 1 - ER finale

Gioele Sforza
Simone Dongiovanni
02/11/2016

2.8 SQL per manipolare i dati

In SQL esistono 3 comandi che possono essere usati per modificare i dati contenuti all'interno di una tabella: INSERT, DELETE, e UPDATE.

- **Query di tipo INSERT**

Nella sua forma più semplice, INSERT è usato per aggiungere una singola tupla (riga) in una relazione (tabella). È necessario specificare il nome della relazione e una lista di valori per la tupla. I valori devono essere specificati nello stesso ordine con cui sono stati specificati i relativi attributi nella query CREATE TABLE.

```
U1:  INSERT INTO EMPLOYEE
      VALUES ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
              Oak Forest, Katy, TX', 'M', 37000, '653298653', 4);
```

Figura 2.73: SQL INSERT

Una seconda formulazione per INSERT consente di specificare esplicitamente gli attributi che corrispondono ai valori che si vogliono inserire:

```

U1A: INSERT INTO EMPLOYEE (Fname, Lname, Dno, Ssn)
      VALUES ('Richard', 'Marini', 4, '653298653');

```

Figura 2.74: SQL INSERT con specifica esplicita degli attributi da inserire

Gli attributi che non sono specificati in questa variante vengono settati al loro valore di default o a NULL. È importante che vengano sempre incluse le chiavi primarie e i campi che non possono assumere valore nullo. **ATTENZIONE!** Se si vuole modificare il valore contenuto all'interno di una riga, non si deve usare la INSERT specificando gli argomenti, ma bisogna usare una query di tipo UPDATE. Infatti, le INSERT creano sempre una nuova riga ogni volta che vengono eseguite.

- **Query di tipo DELETE**

Il comando DELETE rimuove tuple da una relazione. Specificando la condizione WHERE (che funziona come nelle query SELECT) è possibile selezionare quali tuple eliminare. Se invece non viene specificata, la DELETE procederà alla cancellazione di tutte le tuple contenute all'interno della relazione.

```

U4A: DELETE FROM EMPLOYEE
      WHERE Lname = 'Brown';
U4B: DELETE FROM EMPLOYEE
      WHERE Ssn = '123456789';
U4C: DELETE FROM EMPLOYEE
      WHERE Dno = 5;
U4D: DELETE FROM EMPLOYEE;

```

Figura 2.75: SQL DELETE

Spiegazione delle 4 queries:

- U4A: Elimina dalla relazione EMPLOYEE le tuple aventi l'attributo Lname impostato a 'Brown';
- U4B: Elimina dalla relazione EMPLOYEE le tuple aventi l'attributo Ssn impostato a '123456789';
- U4C: Elimina dalla relazione EMPLOYEE le tuple aventi l'attributo Dno impostato a 5;
- U4D: Elimina dalla relazione EMPLOYEE tutte le tuple.

Nonostante questo comando consenta di effettuare delle cancellazioni all'interno del database è importante notare che la legge italiana impone che i dati storici non debbano mai essere cancellati. Tali dati possono risultare importanti, ad esempio, per successive analisi statistiche di varia natura. In effetti, dato il basso costo che lo storage ha oggi, questo non rappresenta un problema. In ambito bancario, ad esempio, una persona potrebbe chiedere di essere cancellata: in tal caso, l'ente in questione è obbligato a farlo, conservando solo i dati rilevanti (come nome, cognome e il fatto che questa persona ha chiesto di essere cancellata). Ciò permette, in futuro, di non includere nuovamente questa

persona all'interno di campagne promozionali. Gestire queste ed altre necessità è compito del responsabile dei dati (che è una figura necessaria quando le dimensioni del business sono importanti). Inoltre, è opportuno notare che quando si effettua una query DELETE, il DBMS adotta delle strategie che permettono di migliorare le performance. Infatti, le tuple eliminate dalla relazione non sono fisicamente rimosse dal filesystem, ma vengono solo rimosse dall'indice e lasciate sul disco. Questo produce un overhead della tabella che cresce man mano che si eliminano tuple dalla relazione. Solo successivamente è possibile ottimizzare la tabella usando alcuni comandi di amministrazione: nell'esecuzione di questi comandi, la tabella verrà distrutta e immediatamente ripopolata (eliminando dunque l'overhead su disco). È importante infine osservare che, generalmente, viene effettuato un backup dell'intero database ogni giorno (solitamente nelle ore notturne, quando non ci sono molti accessi al DB). Lo schema più usato prevede di effettuare un backup completo la domenica (poco traffico) e dei backup incrementali durante la settimana (molto traffico). Ciò significa che per ripristinare ad esempio il backup del mercoledì, sarà necessario applicare in sequenza i backup di domenica, lunedì, martedì, mercoledì. **File di log / journal**: tengono traccia di tutti gli I/O di un software. Nei database sono usati, tra l'altro, per ricostruire transazioni interrotte.

• Query di tipo UPDATE

Il comando UPDATE consente di modificare il valore degli attributi di una o più tuple selezionate. Come nel comando DELETE, anche qui è possibile inserire la clausola WHERE che permette di selezionare quali tuple modificare. **Nota bene:** Come nel caso della DELETE, non specificare la clausola WHERE equivale a modificare **tutte** le tuple di una tabella.

```

U5:   UPDATE    PROJECT
      SET        Plocation = 'Bellaire', Dnum = 5
      WHERE     Pnumber = 10;

U6:   UPDATE    EMPLOYEE
      SET        Salary = Salary * 1.1
      WHERE     Dno = 5;
  
```

Figura 2.76: SQL UPDATE

Spiegazione delle due query:

- U5: Modifica le tuple della relazione PROJECT aventi Pnumber pari a 10 impostando l'attributo Plocation a 'Bellaire' e l'attributo Dnum a 5;
- U6: Modifica le tuple della relazione EMPLOYEE aventi Dno pari a 5 aumentando del 10% l'attributo Salary.

2.8.1 Transazioni e concorrenza (accenno)

Due feature fondamentali dei database sono le **transazioni** e la **concorrenza**. Le accenniamo brevemente e le riprenderemo in seguito. Partiamo ad un breve esempio. Supponiamo che 10 utenti debbano effettuare un acquisto online di un dato prodotto e che siano disponibili solo 5 esemplari di quel prodotto. Supponiamo che solo 5 utenti su 10 abbiano la reale intenzione

di acquistare il prodotto e che gli altri 5 decidano di rinunciare all'acquisto nella fase finale della procedura. Un classico problema posto da questo esempio che ricorre quando si modellano database per l'acquisto online è il seguente: come gestire il blocco del prodotto? Se si opta per bloccare il prodotto una volta iniziata la procedura d'acquisto potrebbe accadere che i 5 utenti realmente interessati non possano acquistarla a causa dei 5 utenti non interessati che rinunceranno al prodotto solo al termine della loro procedura. Questo arrecherebbe danno al venditore poiché 5 persone realmente intenzionate all'acquisto non trovano disponibilità per il prodotto, mentre altre 5 vi rinunciano all'ultimo momento, e la merce resterebbe invenduta. Se si opta per bloccare il prodotto nell'ultima fase della procedura d'acquisto, la merce potrebbe essere stata acquistata da un altro utente pochi istanti prima di terminare l'acquisto e in tal caso non sarebbe possibile completare l'acquisto. Questo porterebbe l'utente a non essere soddisfatto del sito e quindi anche in questo caso ci sarebbe un danno per il venditore. In situazioni come questa la scelta migliore è quella di usare tecniche statistiche, con le quali si blocca ogni minuto il numero medio di prodotti acquistati al minuto. Se ad esempio un'attività vende un certo prodotto 5 volte al minuto, il software deciderà di bloccare ogni minuto una quantità pari a 5 per quel prodotto. Chiaramente, questo tipo di approccio è possibile solo nel caso in cui il business è sufficientemente grande da poter usare la statistica. Caratteristiche importanti che un database deve possedere sono riassunte nell'acronimo **ACID** (Atomicità, Consistenza, Isolamento e Durabilità). Per arrivare ad ottenere un database con queste caratteristiche si sfruttano tecniche matematiche. I database relazionali hanno il vantaggio di godere della proprietà ACID, e hanno quindi la caratteristica di non essere penetrabili. Tale pregio lo si paga in termini di scalabilità e velocità: usando architetture parallele e tracciando lo speedup in funzione dei processori, ad un certo punto la velocità di aumento dello speedup diventa sublineare. I database NoSQL (Not only SQL) non godono generalmente della proprietà ACID e sono invece più performanti su larga scala, ma accettano come ipotesi di perdere qualche informazione. Pertanto questi database perdono in coerenza. La bravura del progettista deve essere quella di conoscere tutte le soluzioni possibili e di adattarle in base alle necessità poste dal problema. In una applicazione bancaria, ad esempio, potrebbe essere un grosso errore quello di adottare una soluzione NoSQL.

2.8.2 Algebra a tre livelli

Nell'ambito del linguaggio SQL è stato necessario ridefinire l'algebra booleana, che generalmente consta di 2 valori (vero e falso), rendendola a tre livelli (vero, falso, null). Sono stati attribuiti 12 possibili significati differenti a NULL, tre dei quali sono:

- *Valore sconosciuto* (ad esempio quando non si conosce la data di nascita di una persona);
- *Valore non applicabile* (ad esempio un attributo LivelloLaurea è NULL per chi non è laureato);
- *Valore non disponibile/trattenuto* (ad esempio se una persona non vuole fornire il proprio indirizzo).

A causa della presenza del valore NULL, le operazioni booleane sono state estese in modo naturale come riassunto in figura:

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN
OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
NOT			
TRUE	FALSE		
FALSE	TRUE		
UNKNOWN	UNKNOWN		

Figura 2.77: Algebra a tre livelli

Se in una query ci fosse la necessità di fare confronti contenenti NULL, non è possibile utilizzare le operazioni di confronto classiche come $\{=, <>, >, <, \geq, \leq\}$, ma si possono usare gli operatori **IS / IS NOT**, poiché l'uguaglianza tra NULL non esiste. Ad esempio, la query seguente seleziona i nomi degli impiegati senza supervisori:

```
Q18:   SELECT      Fname, Lname
        FROM       EMPLOYEE
        WHERE      Super_ssn IS NULL;
```

Figura 2.78: Esempio di utilizzo della clausola IS in SQL

2.8.3 Query più complesse

- **Query annidate**

Le **query annidate** sono delle query di tipo select-from-where all'interno di una **query esterna**. Le query annidate possono comparire in qualunque clausola si renda necessario inserirle: possono essere contenute nel blocco FROM, o in quello WHERE, o anche in quello SELECT. Nella seguente query introduciamo l'operatore di confronto **IN**, che confronta un valore v con un insieme e restituisce TRUE se v è contenuto nell'insieme.

```
Q4A:   SELECT      DISTINCT Pnumber
        FROM       PROJECT
        WHERE      Pnumber IN
                  (SELECT      Pnumber
                   FROM       PROJECT, DEPARTMENT, EMPLOYEE
                   WHERE      Dnum = Dnumber AND
                             Mgr_ssn = Ssn AND Lname = 'Smith' )
                  OR
                  Pnumber IN
                  (SELECT      Pno
                   FROM       WORKS_ON, EMPLOYEE
                   WHERE      Essn = Ssn AND Lname = 'Smith' );
```

Figura 2.79: Query annidata con clausola IN

La query Q4A seleziona i numeri di progetto (distinti) il cui manager o gli impiegati che ci lavorano hanno cognome Smith. Le due query annidate selezionano rispettivamente i numeri di progetto il cui manager ha cognome Smith e i numeri di progetto su cui lavorano impiegati aventi il cognome Smith. Successivamente, la query esterna mette insieme i risultati delle due query interne e restituisce solo i valori distinti (DISTINCT). Se una query annidata restituisce un singolo attributo e un singolo valore (cioè una tabella con una sola riga e una sola colonna), il risultato della query è uno **scalare**. In tal caso, è possibile usare l'operatore = per il confronto. In generale, il risultato di una query annidata è una **tabella** (una relazione con in generale più di una riga e più di una colonna). SQL permette di usare dei confronti tra **tuple** di valori, mettendole tra parentesi. I confronti tra tuple funzionano come i confronti tra vettori. Due tuple si considerano uguali se e solo se tutti gli elementi corrispondenti delle due tuple sono uguali (ad esempio $[1, 2, 3] = [1, 2, 3]$). Due tuple si considerano diverse se e solo se c'è almeno un elemento corrispondente tra le due tuple che è diverso (ad esempio $[1, 2, 3] <> [1, 1, 3]$).

```
SELECT      DISTINCT Essn
FROM        WORKS_ON
WHERE       (Pno, Hours) IN (SELECT      Pno, Hours
                           FROM        WORKS_ON
                           WHERE       Essn = '123456789');
```

Figura 2.80: Query annidata

La query riportata sopra seleziona gli Essn degli impiegati che lavorano allo stesso progetto e per lo stesso numero di ore dell'impiegato con Essn 123456789. È opportuno notare che quando si vogliono fare i confronti tra tuple è necessario che la tuple da confrontare abbiano lo stesso numero di elementi (ovvero che il numero di elementi tra parentesi a sinistra dell'operatore IN coincida col numero di attributi indicati nella SELECT della query annidata). Ci sono altri operatori che si possono usare per confrontare un valore v con un insieme (che è il caso delle query annidate). Query operatori sono ANY, SOME, ALL (SOME è sinonimo di ANY) e si applicano insieme ad un operatore di confronto come quelli visti finora $\{=, \leq, <, >, \geq\}$. Ad esempio, $= \text{ANY}$ ($\text{o} = \text{SOME}$) verifica che l'elemento v sia uguale **ad almeno un elemento** dell'insieme (e quindi è equivalente a IN). Invece, $> \text{ANY}$ verifica che l'elemento v sia maggiore di almeno un elemento dell'insieme. Invece, $> \text{ALL}$ verifica che l'elemento v sia maggiore **di tutti gli elementi** dell'insieme. La seguente query restituisce il nome degli impiegati con salario maggiore di tutti gli impiegati del dipartimento 5:

```
SELECT      Lname, Fname
FROM        EMPLOYEE
WHERE       Salary > ALL (SELECT      Salary
                           FROM        EMPLOYEE
                           WHERE       Dno = 5 );
```

Figura 2.81: Query annidata con clausola ALL

- **Query annidate correlate**

È possibile che una query annidata contenga un riferimento ad un attributo gestito dalla query esterna. In questo caso si dice che le due query sono **correlate**:

- Le query annidate standard vengono valutate una volta. Il risultato è usato poi dalla query esterna;
- Le query annidate correlate vengono valutate una volta per ogni tupla (o combinazione di tuple) nella query esterna. Ad esempio, se voglio trovare le persone che guadagnano più dei loro parenti, non posso calcolare l'insieme dei parenti una volta per tutte, ma devo rifarlo per ogni persona.

Questa query restituisce i nomi degli impiegati che hanno dei dipendenti con il loro stesso nome e sesso:

```
Q16:   SELECT      E.Fname, E.Lname
          FROM        EMPLOYEE AS E
          WHERE       E.Ssn IN  ( SELECT      D.Essn
                                FROM        DEPENDENT AS D
                                WHERE       E.Fname = D.Dependent_name
                                AND E.Sex = D.Sex );
```

Figura 2.82: Query annidata correlata

Spiegazione: Per ogni tupla di EMPLOYEE, viene valutata la query interna, che trova il Ssn dei dipendenti con il loro stesso nome e sesso. Successivamente, per ognuno di questi Ssn viene restituito il nome e il cognome. Il ragionamento viene ripetuto su ogni tupla di EMPLOYEE. È importante quando ci sono query annidate correlate che non ci sia ambiguità negli attributi, cioè non ci possono essere attributi aventi lo stesso nome nella query interna e in quella esterna. Per risolvere queste situazioni, si rende necessario l'uso degli alias come già visto in precedenza per identificare univocamente ogni oggetto presente nella query. Per quanto riguarda le prestazioni, è immediato comprendere che nel caso di query annidate correlate le performance si degradano molto più velocemente che nelle query annidate standard. Infatti, nelle query annidate standard il tempo di esecuzione è lineare nella dimensione della tabella, mentre nelle query annidate correlate il tempo di esecuzione è quadratico nella dimensione della tabella. Fortunatamente, molte delle query annidate correlate sono riscrivibili come una query non annidata con dei JOIN. Il DBMS ha un ottimizzatore incorporato che provvede ad effettuare questa operazione ogni volta che è possibile (riportando la complessità da quadratica a lineare).

- **Funzioni EXISTS e UNIQUE**

Le funzioni **EXISTS** e **UNIQUE** sono funzioni booleane (restituiscono VERO o FALSO) da usare nelle clausole WHERE. La funzione EXISTS restituisce TRUE quando un insieme è non vuoto (cioè contiene tuple; per verificare invece che un insieme non contenga alcuna tupla è sufficiente anteporre un NOT). La funzione UNIQUE restituisce TRUE quando all'interno di un insieme non ci sono tuple ripetute (per verificare invece che un insieme contenga almeno una tupla ripetuta è sufficiente anteporre un NOT).

```

Q6:   SELECT      Fname, Lname
      FROM       EMPLOYEE
      WHERE      NOT EXISTS ( SELECT      *
                                FROM       DEPENDENT
                                WHERE      Ssn = Essn );

```

Figura 2.83: Query con clausola EXISTS

La precedente query (che contiene una query annidata correlata) seleziona gli impiegati che non hanno dipendenti. Spiegazione: per ogni riga di EMPLOYEE, viene eseguita la query interna che seleziona i dipendenti dell'impiegato corrente. Se il risultato è vuoto, l'impiegato corrente viene aggiunto al risultato.

- **Query con JOIN**

In SQL è possibile effettuare il join tra due o più tabelle nella clausola FROM, specificando eventualmente la condizione di join e un alias per le tabelle. In SQL ci sono i seguenti tipi di join (che di default sono INNER JOIN):

- A **JOIN** B ON A.att1 = B.att2: effettua l'EQUIJOIN tra le tabelle A e B, con condizione di join A.att1 = B.att2. Se si vuole effettuare un THETA JOIN, basta usare un operatore di confronto diverso da =;
- A **NATURAL JOIN** B: effettua il NATURAL JOIN tra le tabelle A e B con condizione di join implicita, ovvero un EQUIJOIN per ogni coppia di attributi delle due tabelle aventi lo stesso nome.

È poi possibile effettuare degli OUTER JOIN specificando il tipo di join esterno da applicare (LEFT, RIGHT, FULL) e la keyword OUTER (opzionale). Questi join possono anche essere di tipo NATURAL:

- A **(NATURAL) LEFT (OUTER) JOIN** B **(ON A.att1 = B.att2)**: effettua il join esterno sinistro tra A e B, eventualmente specificando la condizione di join o la keyword NATURAL;
- A **(NATURAL) RIGHT (OUTER) JOIN** B **(ON A.att1 = B.att2)**: effettua il join esterno destro tra A e B, eventualmente specificando la condizione di join o la keyword NATURAL;
- A **(NATURAL) FULL (OUTER) JOIN** B **(ON A.att1 = B.att2)**: effettua il join esterno completo tra A e B, eventualmente specificando la condizione di join o la keyword NATURAL.

- **Funzioni di aggregazione**

Le funzioni aggregazione riassumono le informazioni di molte tuple **in una singola tupla**. È possibile raggruppare le tuple prima di applicare le funzioni di aggregazione usando le keyword apposite. Esempi di funzioni di aggregazione: COUNT (conta il numero di tuple), SUM (somma i valori di tuple diverse), MAX (estrae il massimo da più tuple), MIN (estrae il minimo da più tuple), AVG (calcola la media di valori di tuple diverse), etc. Queste funzioni si possono usare nelle clausole SELECT e HAVING (più avanti). Si può usare la clausola WHERE per prefiltrare le tuple prima di applicare le funzioni (filtraggio a priori).

```
Q19:   SELECT      SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)  
        FROM          EMPLOYEE;
```

Figura 2.84: Query 19: trova la somma, il massimo, il minimo e la media degli stipendi dei dipendenti

```
Q20:   SELECT      SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)  
        FROM          (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)  
        WHERE         Dname = 'Research';
```

Figura 2.85: Query 20: query 1 ristretta al dipartimento "Research":

```
Q21:   SELECT      COUNT (*)  
        FROM          EMPLOYEE;
```

```
Q22:   SELECT      COUNT (*)  
        FROM          EMPLOYEE, DEPARTMENT  
        WHERE         DNO = DNUMBER AND DNAME = 'Research';
```

Figura 2.86: Query 21 e 22: restituisce il numero di impiegati (Q21) e il numero di impiegati nel dipartimento Research (Q22)

- **Raggruppamento e filtraggio nelle funzioni di aggregazione**

Come già accennato, è possibile suddividere le tuple in sottogruppi prima di applicare le funzioni di aggregazione. Se ad esempio un certo raggruppamento produce 3 gruppi, dopo l'applicazione di una funzione di aggregazione si otterranno 3 tuple. La keyword che permette di fare i raggruppamenti è **GROUP BY**, seguita dall'attributo secondo il quale effettuare il raggruppamento. Nella seguente query vengono calcolati il numero di impiegati e il salario medio di ogni dipartimento.

```
Q24:   SELECT      Dno, COUNT (*), AVG (Salary)  
        FROM          EMPLOYEE  
        GROUP BY    Dno;
```

Figura 2.87: Raggruppamento su funzioni di aggregazione mediante GROUP BY

Per filtrare i risultati ottenuti **dopo** l'applicazione di una funzione di aggregazione, bisogna usare la keyword **HAVING** dopo la clausola GROUP BY (se presente). La clausola HAVING permette di specificare delle condizioni sugli attributi aggregati per selezionare alcune tuple. È come il WHERE, ma contiene al suo interno condizioni sugli attributi ottenuti tramite funzioni di aggregazione, e inoltre viene applicato dopo che sono stati

calcolati i valori delle funzioni di aggregazione (filtraggio a posteriori). Nella seguente query, per ogni progetto su cui lavorano più di 2 impiegati vengono selezionati il nome del progetto e il numero di impiegati.

```
Q26:   SELECT      Pnumber, Pname, COUNT (*)
          FROM        PROJECT, WORKS_ON
          WHERE       Pnumber = Pno
          GROUP BY    Pnumber, Pname
          HAVING     COUNT (*) > 2;
```

Figura 2.88: Filtraggio post funzioni di aggregazione mediante HAVING

Riassumendo: mentre la WHERE consente di selezionare le *tuple* su cui vengono applicate le funzioni di aggregazione, la HAVING permette di scegliere *interi gruppi*.

- **Struttura generale di una query SELECT**

```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

Figura 2.89: Struttura generale SELECT

2.8.4 Vincoli in SQL

In SQL esistono diversi tipi di vincoli che è possibile specificare (programmazione event-driven nei database). Abbiamo già analizzato la clausola CHECK all'interno di una CREATE TABLE. Ora analizziamo le asserzioni e i trigger, che sono degli strumenti che permettono di impostare dei vincoli sul contenuto di un database.

2.8.5 Asserzioni

Le asserzioni sono dei vincoli generici dichiarabili con lo statement CREATE ASSERTION. Ogni asserzione ha un nome ed è specificata in modo simile a come si specifica la clausola WHERE di una query di tipo SELECT. La seguente query impone che lo stipendio di un impiegato non può essere più alto di quello del manager del dipartimento per cui lavora.

```

CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT *
                      FROM      EMPLOYEE E, EMPLOYEE M,
                                DEPARTMENT D
                     WHERE    E.Salary > M.Salary
                            AND   E.Dno = D.Dnumber
                            AND   D.Mgr_ssn = M.Ssn ) );

```

Figura 2.90: Statement CREATE ASSERTION e CHECK

Quando verrà inserita una nuova riga nel database, verrà effettuato questo controllo, e verrà restituito un errore in caso di violazione.

2.8.6 Trigger

I trigger sono una generalizzazione dei check e delle asserzioni. Sono dei vincoli semantici di qualunque natura su qualunque oggetto del database. Un trigger è completamente definito da 3 elementi:

- Evento;
- Condizione;
- Azione

Quando, in seguito all'evento, la condizione risulta verificata, viene eseguita l'azione specificata.

2.8.7 Viste (Views)

Le **Views** sono delle tabelle virtuali che è possibile creare all'interno del proprio database. Quando effettuiamo una SELECT, in effetti il DBMS ci restituisce una tabella virtuale, risultato di una serie di operazioni effettuate sulle tabelle realmente memorizzate nel database. Le Views (chiamate anche **prepared statements**) permettono di dare un nome ad una query SELECT che si intende effettuare spesso. A seguito della definizione di una view, ci sarà nel database una nuova tabella virtuale con il nome specificato il cui contenuto è dato dal risultato della query associata. Questa tabella virtuale può essere a sua volta usata nelle altre query che si possono eseguire sul database. **Vantaggi:** una volta che la view viene dichiarata, la query viene immediatamente compilata dal DBMS, in modo tale che la sua esecuzione risulti più veloce quando questa sarà richiamata. Infatti, le query che si eseguono normalmente sul database devono essere tutte preventivamente compilate. Per creare una view, è sufficiente eseguire il comando **CREATE VIEW AS [QUERY]**. La seguente query crea una view chiamata WORKS_ON1, ottenuta mediante una query SELECT che preleva la lista degli impiegati, con i progetti a cui essi lavorano e il numero di ore.

```

V1:   CREATE VIEW    WORKS_ON1
      AS SELECT      Fname, Lname, Pname, Hours
                     FROM      EMPLOYEE, PROJECT, WORKS_ON
                     WHERE     Ssn = Essn AND Pno = Pnumber;

```

Figura 2.91: Creazione di una VIEW

2.8.8 Esercizio

Consideriamo il database più semplice che esista: una entità e una relazione:

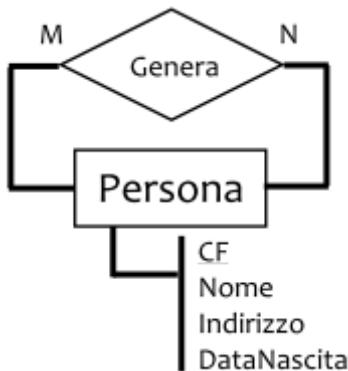


Figura 2.92: Persona Genera Persona

L'algoritmo di mapping genera le seguenti tabelle:



Figura 2.93: Persona Genera Persona - Logical

- Dato il nome del genitore, trovare i nomi dei figli:

```

1 SELECT F.Nome
2 FROM Persona G JOIN Genera G1 ON G.CF = G1.CF_GEN JOIN Persona F
   ON F.CF = G1.CF_DISC
3 WHERE G.Nome = "Nome"
  
```

- Data una persona, trovare i cugini:

```

1 SELECT C.Nome
2 FROM Persona F, Genera G1, Persona P, Genera G2, Persona N, Genera
      G3, Persona Z, Genera G4, Persona C
3 WHERE F.CF = G1.CF_DISC AND G1.CF_GEN = P.CF AND P.CF = G2.CF_DISC
      AND G2.CF_GEN = N.CF AND N.CF = G3.CF_GEN AND G3.CF_DISC = Z
      .CF AND Z.CF = G4.CF_GEN AND G4.CF_DISC = C.CF AND Z.
      CF  $\diamond$  P.CF AND F.Nome = "Nome"
  
```

La clausola aggiuntiva Z.CF \leftrightarrow P.CF permette di ottenere i figli dei figli del nonno o nonna, escluso il padre o madre (ovvero i figli degli zii, cioè i cugini). È possibile riscrivere la query precedenti come una query esterna con una query annidata (non correlata):

```

1 SELECT C.Nome
2 FROM Persona Z           JOIN Genera G4 ON Z.CF = G4.CF_GEN           JOIN
   Persona C ON G4.CF_DISC = C.CF
3 WHERE Z.CF IN
4                               (SELECT G3.CF
5                               FROM Persona F, Genera C1, Persona P, Genera
         G2, Persona N,          Genera G3
6                               WHERE F.CF = G1.CF_DISC AND G1.CF_GEN = P.CF
                               AND          P.CF = G2.CF_DISC AND G2.CF_GEN
                               = N.CF AND          N.CF = G3.CF_GEN AND Z.
                               CF  $\leftrightarrow$  P.CF AND F.Nome = "Nome" )

```

Andrea Camisa
Emanuele Trono
03/11/2016

2.9 Continuazione Popolamento DB

[... continuazione dell'esercizio della lezione precedente]

Nome	Cognome	Provincia	Città	Sex	Age
1 carminelli	Lombardo	EMR	AO	Alfiano N	1502
2 leonzio	Mariani	PUG	TE	Quinto V	1303
3 mariagraz	Rizzo	CAM	AO	Pignataro	304
4 olghina	Moro	BAS	TP	Sant'Elia a	8604
5 terzia	Galli	MAR	RN	Canterano	21
6 manlia	Conti	SIC	Incolla		5703
7 dovgiglia	Fumagalli	VEN			8002
8 celsina	Ferrante	CAM			1502
9 tarquinia	Basile	TAA			2385
0 urano	Gatti	BAS			2503
1 annalisa	Rossi	EMR			6604
2 foscherini	Ferrero	MOL			3602
3 diamante	Pozzi	LAZ			1007
4 filoteo	Bianchi	EMR			6401
5 cleto	Santoro	VEN			1001
6 mirella	Conti	CAL	Incolla speciale...		1002

Figura 2.94: Microsoft Excel - Incolla speciale: Valori

Creiamo un file excel e incolliamo (incolla speciale: incolla valori) i valori copiati dal vecchio file excel e salviamo in formato *.csv* (valori separati da virgola).

Ci sono due metodi per importarlo: o si usa la tabella esistente o si importano i dati dal database, creando una tabella avente come intestazioni la prima riga del file *.csv*.

Quindi andiamo sul nostro database *clienteacquistaprodotto*, “importa” e selezioniamo il file *.csv*:



Figura 2.95: phpMyAdmin - Importazione DB clienteacquistaprodotto

Successivamente selezioniamo il formato *.csv* e osserviamo, con un editor di testo, il modo in cui sono terminati i campi: se con “,” o “;” modificando di conseguenza.

```
ID;Name;Surname;Region;Province;City;CAP;Street;Number;Birthdate
1;firmano;Mariani;VEN;CE;Besate;20080;Moretti;4;31/10/1962
2;cassio;Antonelli;CAM;RC;Cappella Cantone;26020;Baldi;2;03/10/1984
```

Figura 2.96: Verifica CSV

Format:

CSV

Note: If the file contains multiple tables, they will be combined into one.

Format-specific options:

Update data when duplicate keys found on import (add ON DUPLICATE KEY UPDATE)

Columns separated with:

Columns enclosed with:

Columns escaped with:

Lines terminated with:

The first line of the file contains the table column names (if this is unchecked, the first line will become part of the data)

Do not abort on INSERT error

Go

Figura 2.97: Importazione CSV

In questo modo abbiamo creato una tabella “TABLE 5”:

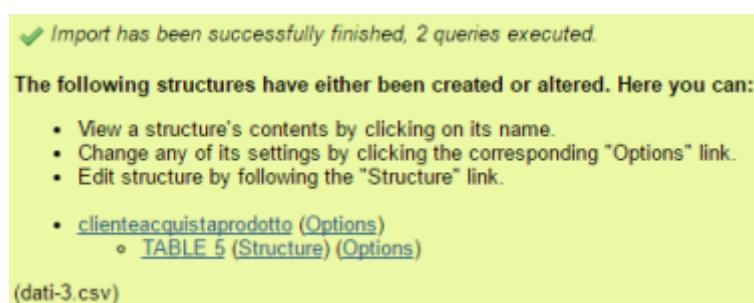


Figura 2.98: Importazione completata con successo

Per cambiare il nome possiamo andare su “Operazioni”, e in “Rinomina tabella” scrivere, ad esempio, “Person”.

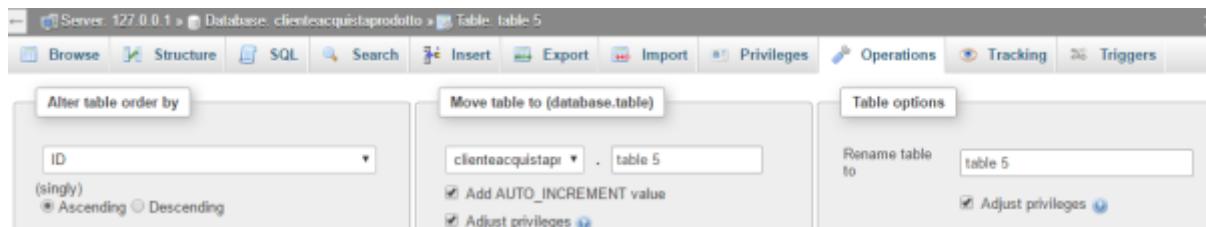


Figura 2.99: Rinominazione tabella

2.9.1 CRUD (Create, Read, Update, Delete)

ARCHITETTURA

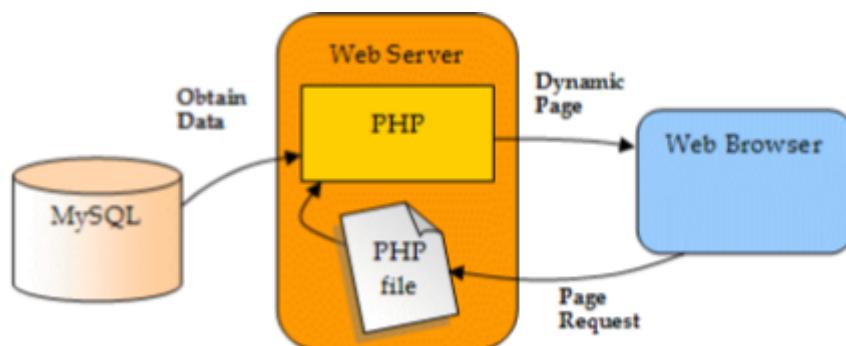


Figura 2.100: Architettura DB - Web Server - Web Browser

Il web browser invia la richiesta di una pagina (un file PHP) al server, che è diviso in Web Server e DBMS, quest'ultimo la crea dinamicamente e la restituisce al client che ne aveva fatto richiesta. Se la pagina è già presente nella cache la prende direttamente dal server, altrimenti comunica con il server MySQL per recuperare i dati e restituirla.

CONNESSIONE AL DB

Comunemente, per effettuare la connessione al DB, viene utilizzata una classe o una libreria, PDO (PHP DATA OBJECTS), un'interfaccia che mette a disposizione i metodi utilizzabili indipendentemente dal DBMS di riferimento. Fornisce un Data-Access Abstraction Layer, cioè un livello di astrazione per l'accesso ai dati. Se PDO non è abilitato: aprire il file di configurazione php.ini e decommentare:

- → extension=php_pdo.dll

Per tutti gli altri tipi di DBMS si va a decommentare le righe relative:

- extension=php_pdo_sqlite.dll
- extension=php_pdo_firebird.dll
- extension=php_mssql.dll

- extension=php_pdo_mysql.dll
- extension=php_pdo_oci.dll
- extension=php_pdo_ibm.dll
- extension=php_pdo_informix.dll
- extension=php_pdo_oci8.dll
- extension=php_pdo_odbc.dll
- extension=php_pdo_pysql.dll

CLASSE PHP PER LA CONNESSIONE AL DB

```

1● class Database {
2      private static $dbName = 'ClienteAcquistaProdotto';
3      private static $dbHost = 'localhost';
4      private static $dbUsername = 'crud';
5      private static $dbUserPassword = 'crud';
6      ...
7 }

```

Gli attributi della classe saranno: il nome, l'host, l'username e password.

```

1● public function __construct() {
2     die('Init function is not allowed');
3 }

```

Questo è il costruttore della classe, che nel nostro caso non lo utilizzeremo. Però visto che è una classe statica, lo dobbiamo inizializzare. Per fare in modo che l'utente non vada ad utilizzare questo nome per un'altra classe ci mettiamo un *die*.

```

1● public static function connect() {
2     if ( null == self::$cont ) {
3         try {
4             self::$cont = new PDO( "mysql:host=".self::
5                         $dbHost . ";" . "dbname=".self::$dbName,
6                         self::$dbUsername, self
7                         ::$dbUserPassword );
8         }
9         catch(PDOException $e) {
10             die($e->getMessage());
11         }
12     }
13     return self::$cont;
14 }

```

La *funzione di connessione* è la funzione principale: usa il pattern Singleton per dire che ci deve essere soltanto la connessione PDO per tutta l'intera connessione. Il blocco *try catch* controlla se l'oggetto di tipo PDO è istanziato (va avanti), altrimenti *die* (restituisce un messaggio di errore).

```

1● public static function disconnect() {
2     self::$cont = null;
3 }

```

La funzione `disconnect` chiude semplicemente la connessione. Bisogna farlo ogni qualvolta si effettua una query.

GRID PER LE OPERAZIONI CRUD

Per creare la nostra applicazione web possiamo usare uno dei tanti strumenti liberi già a disposizione, che ci consentono la creazione di siti e applicazioni, come **Bootstrap**. Questo contiene modelli di progettazione basati su HTML e CSS, sia per la tipografia, che per le varie componenti dell'interfaccia, come moduli, pulsanti e navigazione, così come alcune estensioni opzionali di JavaScript. Una caratteristica molto importante di **Bootstrap** è il fatto che sia *responsivo* con tutti i tipi di dispositivi, inoltre include le *grid table*, che sono molto utilizzate. Per il nostro progetto:

- creiamo il progetto PHP utilizzando uno tra Eclipse, NetBeans, PhpStorm ecc. e lo chiamiamo **MYCRUD**;
- scarichiamo **Bootstrap ver 3.3.7** dal sito ufficiale:
<http://getbootstrap.com/gettingstarted/#download>;
- creiamo 4 pagine PHP, ognuna per un'operazione CRUD che lasceremo per il momento vuote: **{create.php, read.php, update.php, delete.php}**;
- creiamo un file **database.php** per la connessione al database;
- creiamo il file **index.php** che contiene la griglia *Bootstrap*;
- copiamo le cartelle **css, fonts e js** presenti nel *Bootstrap* scaricato, nel nostro progetto.

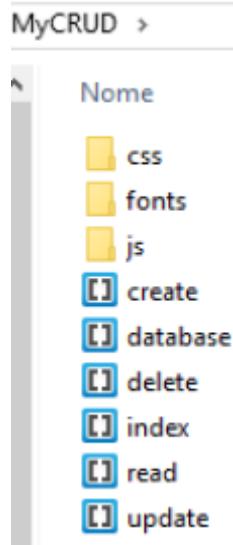


Figura 2.101: Configurazione file/cartelle di Bootstrap per il nostro progetto

È importante che il progetto si trovi nella cartella htdocs di xampp
Scriviamo **index.php**:

```

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link href="css/bootstrap.min.css" rel="stylesheet">
        <script src="js/bootstrap.min.js"></script>
    </head>
    <body>
        <div class="container">
            <div class="row">
                <h3>My PHP CRUD</h3>
            </div>
            <div class="row">
                <table class="table table-striped table-bordered">
                    <thead>
                        <tr>
                            <th>Name</th>
                            <th>Surname</th>
                            <th>City</th>
                            <th>Birthdate</th>
                            <th>Action</th>
                        </tr>
                    </thead>
                    <tbody>
                        <?php
                            include'database.php';
                            $pdo = Database::connect();
                            $sql = 'SELECT * FROM Person ORDER BY ID DESC';
                            foreach ($pdo->query($sql) as $row) {
                                echo '<tr>';
                                echo '<td>'. $row['Name'] . '</td>';
                                echo '<td>'. $row['Surname'] . '</td>';
                                echo '<td>'. $row['City'] . '</td>';
                                echo '<td>'. $row['Birthdate']. '</td>';
                                echo '</tr>';
                            }
                            Database::disconnect();
                        ?>
                    </tbody>
                </table>
            </div>
        </div>
    </body>
</html>

```

Figura 2.102: Configurazione file/cartelle di Bootstrap per il nostro progetto

Contiene un titolo e la *grid* di *Bootstrap* che per adesso è vuota. Ricordiamo di inserire il tag per l'aggiunta dei fogli di stile (**css**) e il link per il **js**.

```

<link href="css/bootstrap.min.css" rel="stylesheet">
<script src="js/bootstrap.min.js"></script>

```

Figura 2.103: TAG per CSS ed il JS

Il **body** avrà un **container** al cui interno è presente il codice php che effettua la connessione al db, la query per leggere le informazioni dalla tabella *Person* e la chiusura della connessione. Il risultato sarà, per il momento, una tabella vuota perché ancora non è popolata.



Figura 2.104: MyPHP CRUD

Aggiungiamo il bottone **Create** in **index.php**, con il quale potremmo iniziare a popolare la tabella *Person*.

```
<div class="row">
    <p>
        <a href="create.php" class="btn btn-success">Create</a>
    </p>
```

Figura 2.105: Aggiunta bottone Create



Figura 2.106: MyPHP CRUD - Pulsante CREATE

E tutti gli altri bottoni, **Read**, **Update** e **Delete** che effettueranno le altre operazioni CRUD.

```
echo '<a class="btn" href="read.php?id='.$row['ID'].'">Read</a>';
echo '&nbsp;';
echo '<a class="btn btn-success" href="update.php?id='.$row['ID'].'">Update</a>';
echo '&nbsp;';
echo '<a class="btn btn-danger" href="delete.php?id='.$row['ID'].'">Delete</a>';
echo '</td>';
echo '</tr>';
```

Figura 2.107: Aggiunta altri bottoni (C)RUD

I bottoni inseriti *Create*, *Read*, *Update*, *Delete*, fanno riferimento alle quattro pagine php, rispettivamente **create.php**, **read.php**, **update.php** e **delete.php**. Pertanto per effettuare le operazioni CRUD dobbiamo definire in ciascuno di essi le chiamate da effettuare sul database e quindi le rispettive queries che interrogheranno il database.

- `create.php`:

```

<?php
require 'database.php';

if (!empty($_POST)) {
    // keep track validation errors
    $nameError = null;
    $surnameError = null;
    $cityError = null;
    $birthdateError = null;

    // keep track post values
    $name = $_POST['name'];
    $surname = $_POST['surname'];
    $city = $_POST['city'];
    $birthdate = $_POST['birthdate'];

    // validate input
    $valid = true;
    if (empty($name)) {
        $nameError = 'Please enter Name';
        $valid = false;
    }
    if (empty($surname)) {
        $surnameError = 'Please enter Surname';
        $valid = false;
    }
    if (empty($city)) {
        $cityError = 'Please enter City';
        $valid = false;
    }

    // insert data
    if ($valid) {
        $pdo = Database::connect();
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        $sql = "INSERT INTO person (name,surname,city, birthdate) values(?, ?, ?, ?)";
        $q = $pdo->prepare($sql);
        $q->execute(array($name, $surname, $city, $birthdate));
        Database::disconnect();
        header("Location: index.php");
    }
}
?>

```

Figura 2.108: `create.php`

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <script src="js/bootstrap.min.js"></script>
</head>
<body>
    <div classe="container">
        <div class="span10 offset1">
            <div class="row">
                <h3>Create a new Person</h3>
            </div>
            <form class="form-horizontal" action="create.php" method="post">
                <div class="control-group <?php echo!empty($nameError) ? 'error' : ''; ?>">
                    <label class="control-label">Name</label>
                    <div class="controls">
                        <input name="name" type="text" placeholder="Name" value="<?php echo!empty($name) ? $name : ''; ?>">
                        <?php if (!empty($nameError)) : ?>
                            <span class="help-inline"><?php echo $nameError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($surnameError) ? 'error' : ''; ?>">
                    <label class="control-label">Surname</label>
                    <div class="controls">
                        <input name="surname" type="text" placeholder="Surname" value="<?php echo!empty($surname) ? $surname : ''; ?>">
                        <?php if (!empty($surnameError)) : ?>
                            <span class="help-inline"><?php echo $surnameError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($cityError) ? 'error' : ''; ?>">
                    <label class="control-label">City</label>
                    <div class="controls">
                        <input name="city" type="text" placeholder="City" value="<?php echo!empty($city) ? $city : ''; ?>">
                        <?php if (!empty($cityError)) : ?>
                            <span class="help-inline"><?php echo $cityError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($birthdateError) ? 'error' : ''; ?>">
                    <label class="control-label">Birthdate</label>
                    <div class="controls">
                        <input name="birthdate" type="text" placeholder="Birthdate (yyyy-mm-dd)" value="<?php echo!empty($birthdate) ? $birthdate : ''; ?>">
                        <?php if (!empty($birthdateError)) : ?>
                            <span class="help-inline"><?php echo $birthdateError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="form-actions">
                    <button type="submit" class="btn btn-success">Create</button>
                    <a class="btn" href="index.php">Back</a>
                </div>
            </form>
        </div>
    </div>
</body>
</html>

```

Figura 2.109: create.php - Continuazione

- update.php:

```

<?php
    require 'database.php';
    $id = null;
    if ( !empty($_GET['id'])) {
        $id = $_REQUEST['id'];
    }

    if ( null==$id ) {
        header("Location: index.php");
    }
    if ( !empty($_POST)) {
        // keep track validation errors
        $nameError = null;
        $surnameError = null;
        $cityError = null;
        $birthdateError = null;

        // keep track post values
        $name = $_POST['name'];
        $surname = $_POST['surname'];
        $city = $_POST['city'];
        $birthdate = $_POST['birthdate'];

        // validate input
        $valid = true;
        if (empty($name)) {
            $nameError = 'Please enter Name';
            $valid = false;
        }
        if (empty($surname)) {
            $surnameError = 'Please enter Surname';
            $valid = false;
        }
        if (empty($city)) {
            $cityError = 'Please enter City';
            $valid = false;
        }

        // update data
        if ($valid) {
            $pdo = Database::connect();
            $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            $sql = "UPDATE person set name = ?, surname = ?, city = ?, birthdate = ? WHERE id = ?";
            $q = $pdo->prepare($sql);
            $q->execute(array($name, $surname, $city, $birthdate, $id));
            Database::disconnect();
            header("Location: index.php");
        }
    } else {
        $pdo = Database::connect();
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        $sql = "SELECT * FROM person where id = ?";
        $q = $pdo->prepare($sql);
        $q->execute(array($id));
        $q->fetch(PDO::FETCH_ASSOC);
        $data = $q->fetch(PDO::FETCH_ASSOC);
        $name = $data['Name'];
        $surname = $data['Surname'];
        $city = $data['City'];
        $birthdate = $data['Birthdate'];
        Database::disconnect();
    }
?>

```

Figura 2.110: update.php

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <script src="js/bootstrap.min.js"></script>
</head>
<body>
    <div class="container">
        <div class="span10 offset1">
            <div class="row">
                <h3>Update a Person</h3>
            </div>

            <form class="form-horizontal" action="update.php?id=<?php echo $id ?>" method="post">
                <div class="control-group <?php echo!empty($nameError) ? 'error' : '' ?>">
                    <label class="control-label">Name</label>
                    <div class="controls">
                        <input name="name" type="text" placeholder="Name" value="<?php echo!empty($name) ? $name : '' ?>">
                        <?php if (!empty($nameError)): ?>
                            <span class="help-inline"><?php echo $nameError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($surnameError) ? 'error' : '' ?>">
                    <label class="control-label">Surname</label>
                    <div class="controls">
                        <input name="surname" type="text" placeholder="Surname" value="<?php echo!empty($surname) ? $surname : '' ?>">
                        <?php if (!empty($surnameError)): ?>
                            <span class="help-inline"><?php echo $surnameError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($cityError) ? 'error' : '' ?>">
                    <label class="control-label">City</label>
                    <div class="controls">
                        <input name="city" type="text" placeholder="City" value="<?php echo!empty($city) ? $city : '' ?>">
                        <?php if (!empty($cityError)): ?>
                            <span class="help-inline"><?php echo $cityError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($birthdateError) ? 'error' : '' ?>">
                    <label class="control-label">Birthdate</label>
                    <div class="controls">
                        <input name="birthdate" type="text" placeholder="Birthdate" value="<?php echo!empty($birthdate) ? $birthdate : '' ?>">
                        <?php if (!empty($birthdateError)): ?>
                            <span class="help-inline"><?php echo $birthdateError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="form-actions">
                    <button type="submit" class="btn btn-success">Update</button>
                    <a class="btn" href="index.php">Back</a>
                </div>
            </form>
        </div>
    </div>
</body>
</html>

```

Figura 2.111: update.php - Continuazione

- **read.php:**

```

<?php
require 'database.php';
$id = null;
if (!empty($_GET['id'])) {
    $id = $_REQUEST['id'];
}

if (null == $id) {
    header("Location: index.php");
} else {
    $pdo = Database::connect();
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $sql = "SELECT * FROM person where id = ?";
    $q = $pdo->prepare($sql);
    $q->execute(array($id));
    $data = $q->fetch(PDO::FETCH_ASSOC);
    Database::disconnect();
}
?>

```

Figura 2.112: read.php

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <script src="js/bootstrap.min.js"></script>
</head>
<body>
    <div class="container">
        <div class="span10 offset2">
            <div class="row">
                <h3>Read a Person</h3>
            </div>
            <form class="form-horizontal" method="post">
                <div class="control-group">
                    <label class="control-label">Name</label>
                    <div class="controls">
                        <input name="name" type="text" placeholder="Name" disabled="disabled" value=<?php echo empty($data['Name']) ? $data['Name'] : '' ; ?>>
                    </div>
                </div>
                <div class="control-group">
                    <label class="control-label">Surname</label>
                    <div class="controls">
                        <input name="surname" type="text" placeholder="Surname" disabled="disabled" value=<?php echo empty($data['Surname']) ? $data['Surname'] : '' ; ?>>
                    </div>
                </div>
                <div class="control-group">
                    <label class="control-label">City</label>
                    <div class="controls">
                        <input name="city" type="text" placeholder="City" disabled="disabled" value=<?php echo empty($data['City']) ? $data['City'] : '' ; ?>>
                    </div>
                </div>
                <div class="control-group">
                    <label class="control-label">Birthdate</label>
                    <div class="controls">
                        <input name="birthdate" type="text" placeholder="Birthdate (yyyy-mm-dd)" disabled="disabled" value=<?php echo empty($data['Birthdate']) ? $data['Birthdate'] : '' ; ?>>
                    </div>
                </div>
                <div class="form-actions">
                    <a class="btn" href="index.php">Back</a>
                </div>
            </div>
        </div>
    </div>
</body>
</html>

```

Figura 2.113: read.php - Continuazione

- delete.php:

```

<?php
require 'database.php';
$Id = 0;

if (!empty($_GET['id'])) {
    $Id = $_REQUEST['id'];
}

if (!empty($_POST)) {
    // keep track post values
    $Id = $_POST['id'];

    // delete data
    $pdo = Database::connect();
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $sql = "DELETE FROM person WHERE id = ?";
    $q = $pdo->prepare($sql);
    $q->execute(array($Id));
    Database::disconnect();
    header("Location: index.php");
}
?>

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link href="css/bootstrap.min.css" rel="stylesheet">
        <script src="js/bootstrap.min.js"></script>
    </head>

    <body>
        <div class="container">

            <div class="span10 offset1">
                <div class="row">
                    <h3>Delete a Person</h3>
                </div>

                <form class="form-horizontal" action="delete.php" method="post">
                    <input type="hidden" name="id" value="<?php echo $Id; ?>" />
                    <p class="alert alert-error">Are you sure to delete ?</p>
                    <div class="form-actions">
                        <button type="submit" class="btn btn-danger">Yes</button>
                        <a class="btn" href="index.php">No</a>
                    </div>
                </form>
            </div>
        </div>
    </body>
</html>

```

Figura 2.114: delete.php

Ora per creare un nuovo utente e accedere con le sue credenziali, in PhPMyAdmin:

- **clienteacquistaprodotto** → **Privilegi** → aggiungi account utente;
- inserire nome utente, il localhost e la password;
- selezionare “Garantisce tutti i privilegi per il database ”clienteacquistaprodotto””;
- selezionare “privilegi globali”;

Add user account

Login Information

User name: crud
Host name: localhost
Password: _____
Re-type: _____
Authentication Plugin: Native MySQL authentication
Generate password:

Database for user account

Create database with same name and grant all privileges.
 Grant all privileges on wildcard name (%username%).
 Grant all privileges on database "dermedicinaprodotto".

Global privileges Check all

Note: MySQL privilege names are expressed in English.

<input checked="" type="checkbox"/> Data	<input checked="" type="checkbox"/> Structure	<input checked="" type="checkbox"/> Administration	<input checked="" type="checkbox"/> Resource limits	<input type="checkbox"/> Require SSL
<input checked="" type="checkbox"/> SELECT <input checked="" type="checkbox"/> INSERT <input checked="" type="checkbox"/> UPDATE <input checked="" type="checkbox"/> DELETE <input checked="" type="checkbox"/> FILE	<input checked="" type="checkbox"/> CREATE <input checked="" type="checkbox"/> ALTER <input checked="" type="checkbox"/> INDEX <input checked="" type="checkbox"/> DROP <input checked="" type="checkbox"/> CREATE TEMPORARY TABLES <input checked="" type="checkbox"/> SHOW VIEW <input checked="" type="checkbox"/> CREATE ROUTINE <input checked="" type="checkbox"/> ALTER ROUTINE <input checked="" type="checkbox"/> EXECUTE <input checked="" type="checkbox"/> CREATE VIEW <input checked="" type="checkbox"/> EVENT <input checked="" type="checkbox"/> TRIGGER	<input checked="" type="checkbox"/> GRANT <input checked="" type="checkbox"/> SUPER <input checked="" type="checkbox"/> PROCESS <input checked="" type="checkbox"/> RELOAD <input checked="" type="checkbox"/> SHUTDOWN <input checked="" type="checkbox"/> SHOW DATABASES <input checked="" type="checkbox"/> LOCK TABLES <input checked="" type="checkbox"/> REFERENCES <input checked="" type="checkbox"/> REPLICATION CLIENT <input checked="" type="checkbox"/> REPLICATION SLAVE <input checked="" type="checkbox"/> CREATE USER	<small>Note: Setting these options to 0 (zero) removes the limit.</small> MAX QUERIES PER HOUR: 0 MAX UPDATES PER HOUR: 0 MAX CONNECTIONS PER HOUR: 0 MAX USER CONNECTIONS: 0	<input type="radio"/> SPECIFIED REQUIRE COPIER REQUIRE ISSUER REQUIRE SUBJECT <input type="radio"/> REQUIRE X509 <input type="radio"/> REQUIRE SSL

Figura 2.115: phpmyAdmin - Aggiunta nuovo Account

Per vedere gli utenti associati al database, in PhpMyAdmin:

- server:127.0.0.1 → account utenti.

User accounts overview

User accounts overview

⚠ A user account allowing any user from localhost to connect is present. This will prevent other users from connecting (%) host. [\[?\]](#)

User name	Host name	Password	Global privileges	User group	Grant	Action
<input type="checkbox"/> Any	%	No	USAGE		No	Edit privileges Export
<input type="checkbox"/> Any	localhost	No	USAGE		No	Edit privileges Export
<input type="checkbox"/> crud	localhost	Yes	ALL PRIVILEGES		Yes	Edit privileges Export
<input type="checkbox"/> pma	localhost	No	USAGE		No	Edit privileges Export
<input type="checkbox"/> root	127.0.0.1	No	ALL PRIVILEGES		Yes	Edit privileges Export
<input type="checkbox"/> root	::1	No	ALL PRIVILEGES		Yes	Edit privileges Export
<input type="checkbox"/> root	localhost	No	ALL PRIVILEGES		Yes	Edit privileges Export

Figura 2.116: phpmyAdmin - Riepilogo Utenti

In “modifica privilegi” o (“Edit privileges”) si possono assegnare o togliere alcuni privilegi ad un utente o aggiungere anche altri database su cui l’utente ha quei privilegi, cambiare la password e avere informazioni sul login.

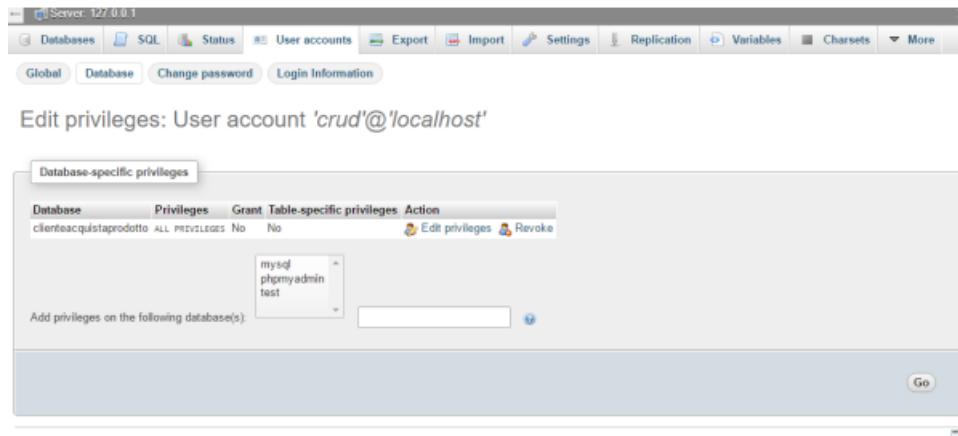


Figura 2.117: phpmyAdmin - Modifica privilegi Utente

Nel file **database.php** modificare la **UserPassword** e lo **Username** con quelli dell’utente appena creato, per connetterci a database con quelle credenziali e avere i privilegi associati a tale utente.

Se le operazioni CRUD non funzionano bisogna rendere l’attributo **id** (della tabella *Person*) autoincrementante, in quanto con l’importazione statica da excel si è posto l’**id** solo come chiave primaria, ma non incrementante. In PhpMyAdmin:

- Database → cliccare su “struttura” sulla riga relativa alla tabella *Person* → cliccare su “change” relativo all’id → spuntare “A_I”

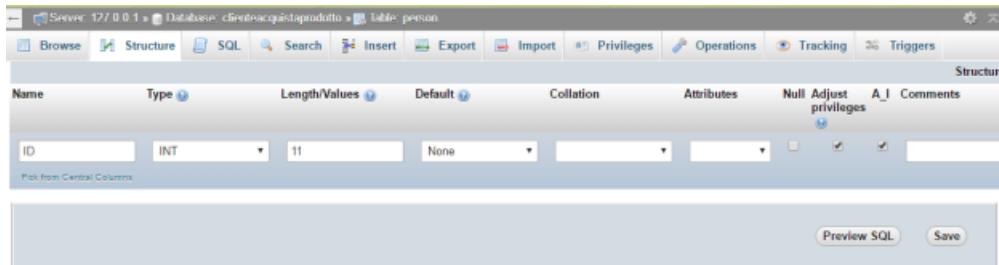


Figura 2.118: phpmyAdmin - Rendere autoincrementante un indice

A questo punto le operazioni CRUD sul database funzionano correttamente:

- Per creare un utente e popolare la tabella *Person*:

The screenshot shows a web browser window with the address bar displaying 'localhost/MYCRUD/create.php'. The main content area is titled 'Create a new Person'. It contains four input fields: 'Name' (with placeholder 'Name'), 'Surname' (with placeholder 'Surname'), 'City' (with placeholder 'City'), and 'Birthdate' (with placeholder 'Birthdate (yyyy-mm-dd)'). Below the inputs are two buttons: a green 'Create' button and a blue 'Back' button.

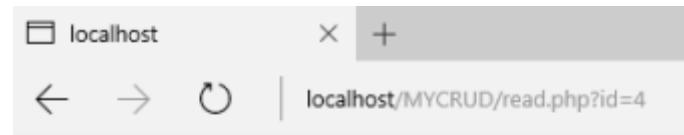
Figura 2.119: PHP CRUD - Creare nuova Persona

- Per modificare un utente

The screenshot shows a web browser window with the address bar displaying 'localhost/MYCRUD/update.php?id=4'. The main content area is titled 'Update a Person'. It contains four input fields: 'Name' (with value 'Mario'), 'Surname' (with value 'Bianchi'), 'City' (with value 'Torino'), and 'Birthdate' (with value '1985-03-20'). Below the inputs are two buttons: a green 'Update' button and a blue 'Back' button.

Figura 2.120: PHP CRUD - Modificare una Persona

- Per leggere le informazioni relative di un utente:



Read a Person

Name

Surname

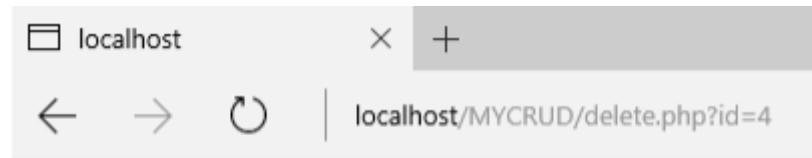
City

Birthdate

[Back](#)

Figura 2.121: PHP CRUD - Leggere informazioni su una Persona

- Per cancellare un utente:



Delete a Person

Are you sure to delete ?

[Yes](#) [No](#)

Figura 2.122: PHP CRUD - Cancellare una Persona

2.10 CRUD cycle

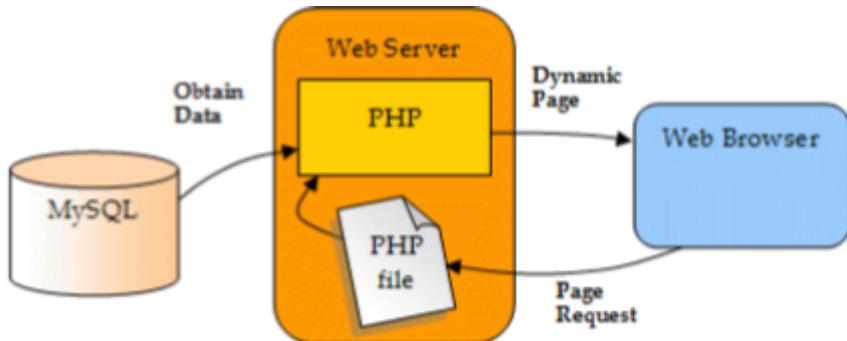


Figura 2.123: Architettura tipica

- Tipicamente i dati sono memorizzati in un DB MySQL;
- PHP è il linguaggio server-side che manipola le tabelle MySQL per consentire all'utente nel front-end di eseguire azioni (CRUD) sui dati.

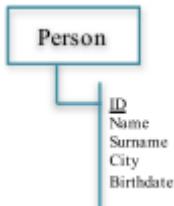


Figura 2.124: Entità Persona

```
1 CREATE TABLE `Person` (
2   `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
3   `Name` VARCHAR(45) NOT NULL ,
4   `Surname` VARCHAR(45) NOT NULL ,
5   `City` VARCHAR(45) NOT NULL,
6   `Birthdate` DATE,
7 ) ENGINE = INNODB;
```

Classe PHP per la connessione al DB:

```
1 <?php class Database {
2     private static $dbName= 'ClienteAcquistaProdotto';
3     private static $dbHost= 'localhost';
4     private static $dbUsername = 'root';
5     private static $dbUserPassword = 'root';
6     private static $cont= null;
7
8     public function __construct() {
9         die('Initfunction is not allowed');
```

```

10 }
11
12     public static function connect() {
13         if( null == self::$cont){
14             try { self::$cont= newPDO("mysql:host=". self::
15                 $dbHost .";" . "dbname=". self::$dbName, self::
16                 $dbUsername , self::$dbUserPassword );
17             }
18             catch(PDOException $e) {
19                 die($e->getMessage());
20             }
21         }
22     }
23     public static function disconnect() {
24         self::$cont= null;
25     }
26 }
27 ?>

```

2.10.1 PDO (PhpData Objects)

- PDO è un'estensione (introdotta nell'implementazione della versione 5 di PHP) che definisce un'interfaccia unica, leggera e consistente per accedere alle basi di dati e che offre allo sviluppatore una classe in grado di fornire metodi utilizzabili indipendentemente dal DBMS di riferimento;
- PDO fornisce un data-access abstraction layer, cioè un livello di astrazione per l'accesso ai dati; si tratta infatti di una classe, definita forse impropriamente anche come "libreria", che mette a disposizione un insieme di sotto-classi derivate che agiscono in modo trasparente rispetto all'utente
- Se PDO non è abilitato: aprire il file di configurazione php.ini e decommentare:

```
extension=php_pdo.dll
```

Figura 2.125: Estensione PDO per PHP

- Decommentare poi le righe relative alle DLL di supporto per i DBMS che si desidera utilizzare:

```

;extension=php_pdo_sqlite.dll
;extension=php_pdo_firebird.dll
;extension=php_pdo_mssql.dll
;extension=php_pdo_mysql.dll
;extension=php_pdo_oci.dll
;extension=php_pdo_ibm.dll
;extension=php_pdo_informix.dll
;extension=php_pdo_ocib.dll
;extension=php_pdo_odbc.dll
;extension=php_pdo_pgsql.dll

```

Figura 2.126: Estensioni PDO per PHP

```

1● public function __construct() {
2      die('Initfunction is not allowed');
3 }

```

Costruttore della classe Database Essendo una classe statica, l'inizializzazione della classe non è consentita. Per impedire l'abuso della classe, inseriamo un die() per ricordarlo all'utente;

```

1● public static function connect() {
2     if( null == self::$cont) {
3         try {
4             self::$cont= newPDO("mysql:host=".self::
5                             $dbHost . ";" . "dbname=". self::$dbName, self
6                             ::$dbUsername , self::$dbUserPassword );
7         }
8         catch(PDOException $e) {
9             die($e->getMessage());
10        }
11    }
12    return self::$cont;
13 }

```

Funzione principale della classe. Usa il pattern Singleton per assicurarsi che esista una sola connessione PDO per l'intera applicazione;

```

1● public static function disconnect() {
2     self::$cont= null;
3 }

```

2.10.2 Grid per le operazioni CRUD



Figura 2.127: Bootstrap from Twitter

- **Bootstrap:** una raccolta di strumenti liberi per la creazione di siti e applicazioni per il Web;

- Contiene modelli di progettazione basati su HTML e CSS, sia per la tipografia, che per le varie componenti dell'interfaccia, come moduli, pulsanti e navigazione, così come alcune estensioni opzionali di JavaScript;
- È compatibile con le ultime versioni di tutti i principali browser;
- Dalla versione 2.0 supporta anche il responsive web design: il layout delle pagine web si regola dinamicamente, tenendo conto delle caratteristiche del dispositivo utilizzato, sia esso desktop, tablet o smartphone;
- A partire dalla versione 3.0, Bootstrap ha adottato il responsive design come impostazione predefinita, sottolineando il suo essere nata come libreria multi dispositivo e multipiattaforma.

Bootstrap

Scarichiamo Bootstrap dal sito ufficiale:
<http://getbootstrap.com/getting-started/#download>

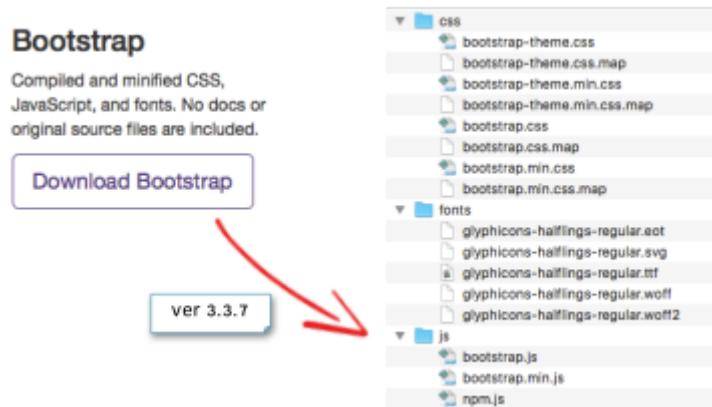


Figura 2.128: Bootstrap - Link ufficiale di download

Nuovo progetto PHP:

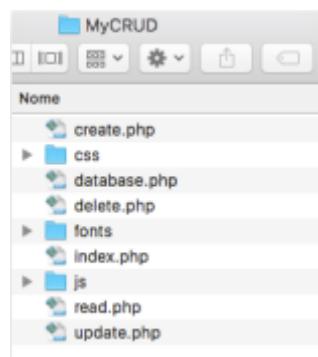


Figura 2.129: Nuovo progetto PHP

Oltre ai file di Bootstrap necessitiamo di:

- 4 file php per le operazioni CRUD (**create.php**, **read.php**, **update.php**, **delete.php**);
- 1 classe **database.php** per la connessione al database (classe Database vista prima);
- 1 file **index.php** che contiene la griglia Bootstrap.

Listiamo l'index.php:

```
1 <!DOCTYPE html>
2 <html lang="en">
3     <head>
4         <meta charset="utf-8">
5         <link href="css/bootstrap.min.css" rel="stylesheet">
6         <script src="js/bootstrap.min.js"></script>
7     </head>
8     <body>
9         <div class="container">
10            <div class="row">
11                <h3>My PHPCRUD</h3>
12            </div>
13            <div class="row">
14                <table class="table table-striped table-bordered">
15                    <thead>
16                        <tr>
17                            <th>Name</th>
18                            <th>Surname</th>
19                            <th>City</th>
20                            <th>Birthdate</th>
21                            <th>Action</th>
22                        </tr>
23                    </thead>
24                    <tbody>
25                        <?php
26                            include 'database .php';
27                            $pdo= Database :: connect ();
28                            $sql= 'SELECT * FROM Person ORDER BY ID DESC';
29                            foreach($pdo->query($sql) as $row) {
30                                echo '<tr>';
31                                echo '<td>' . $row[ 'Name' ] . '</td>';
32                                echo '<td>' . $row[ 'Surname' ] . '</td>';
33                                echo '<td>' . $row[ 'City' ] . '</td>';
34                                echo '<td>' . $row[ 'Birthdate' ] . '</td>';
35                                echo '</tr>';
36                            }
37                            Database :: disconnect ();
38                        ?>
39                    </tbody>
40                </table>
41            </div>
42        </div>
43    </body>
44 </html>
```

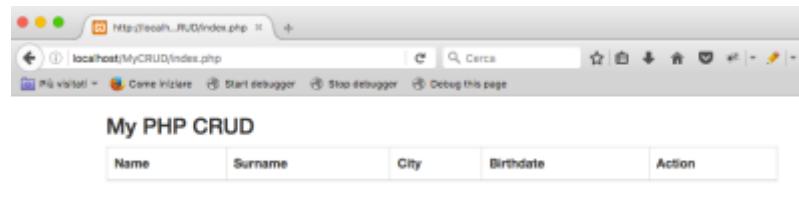


Figura 2.130: index.php

Aggiungiamo alla index il bottone Create...

```

1 ...
2 <div class="row">
3     <p>
4         <a href="create.php" class="btn btn-success">Create</a>
5     </p>
6 ...

```

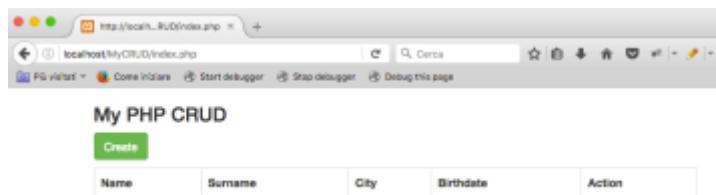


Figura 2.131: index.php con bottone aggiunto

```

1 ...
2 foreach($pdo->query($sql) as $row) {
3     ...
4     echo '<tdwidth=250> ';
5     echo '<a class="btn" href="read.php?id=' . $row[ 'ID' ] . '">Read</a>';
6     echo '&nbsp; ';
7     echo '<a class="btn btn-success" href="update.php?id=' . $row[ 'ID' ] . '">
        Update</a>';
8     echo '&nbsp; ';
9     echo '<a class="btn btn-danger" href="delete.php?id=' . $row[ 'ID' ] . '">
        Delete</a>';
10    echo '</td>';
11    echo '</tr>';
12 }
13 ...

```

- *create.php:*

```

<?php
require 'database.php';

if (!empty($_POST)) {
    // keep track validation errors
    $nameError = null;
    $surnameError = null;
    $cityError = null;
    $birthdateError = null;

    // keep track post values
    $name = $_POST['name'];
    $surname = $_POST['surname'];
    $city = $_POST['city'];
    $birthdate = $_POST['birthdate'];

    // validate input
    $valid = true;
    if (empty($name)) {
        $nameError = 'Please enter Name';
        $valid = false;
    }
    if (empty($surname)) {
        $surnameError = 'Please enter Surname';
        $valid = false;
    }
    if (empty($city)) {
        $cityError = 'Please enter City';
        $valid = false;
    }

    // insert data
    if ($valid) {
        $pdo = Database::connect();
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        $sql = "INSERT INTO person (name,surname,city, birthdate) values(?, ?, ?, ?)";
        $q = $pdo->prepare($sql);
        $q->execute(array($name, $surname, $city, $birthdate));
        Database::disconnect();
        header("Location: index.php");
    }
}
?>

```

Figura 2.132: create.php

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <script src="js/bootstrap.min.js"></script>
</head>
<body>
    <div class="container">
        <div class="span10 offset1">
            <div class="row">
                <h3>Create a new Person</h3>
            </div>
            <form class="form-horizontal" action="create.php" method="post">
                <div class="control-group <?php echo!empty($nameError) ? 'error' : ''; ?>">
                    <label class="control-label">Name</label>
                    <div class="controls">
                        <input name="name" type="text" placeholder="Name" value="<?php echo!empty($name) ? $name : ''; ?>">
                        <?php if (!empty($nameError)): ?>
                            <span class="help-inline"><?php echo $nameError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($surnameError) ? 'error' : ''; ?>">
                    <label class="control-label">Surname</label>
                    <div class="controls">
                        <input name="surname" type="text" placeholder="Surname" value="<?php echo!empty($surname) ? $surname : ''; ?>">
                        <?php if (!empty($surnameError)): ?>
                            <span class="help-inline"><?php echo $surnameError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($cityError) ? 'error' : ''; ?>">
                    <label class="control-label">City</label>
                    <div class="controls">
                        <input name="city" type="text" placeholder="City" value="<?php echo!empty($city) ? $city : ''; ?>">
                        <?php if (!empty($cityError)): ?>
                            <span class="help-inline"><?php echo $cityError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($birthdateError) ? 'error' : ''; ?>">
                    <label class="control-label">Birthdate</label>
                    <div class="controls">
                        <input name="birthdate" type="text" placeholder="Birthdate (yyyy-mm-dd)" value="<?php echo!empty($birthdate) ? $birthdate : ''; ?>">
                        <?php if (!empty($birthdateError)): ?>
                            <span class="help-inline"><?php echo $birthdateError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="form-actions">
                    <button type="submit" class="btn btn-success">Create</button>
                    <a class="btn" href="index.php">Back</a>
                </div>
            </form>
        </div>
    </div>
</body>
</html>

```

Figura 2.133: create.php

- *read.php*:

```

<?php
require 'database.php';
$id = null;
if (!empty($_GET['id'])) {
    $id = $_REQUEST['id'];
}

if (null == $id) {
    header("Location: index.php");
} else {
    $pdo = Database::connect();
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $sql = "SELECT * FROM person where id = ?";
    $q = $pdo->prepare($sql);
    $q->execute(array($id));
    $data = $q->fetch(PDO::FETCH_ASSOC);
    Database::disconnect();
}
?>

```

Figura 2.134: read.php

```

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link href="css/bootstrap.min.css" rel="stylesheet">
        <script src="js/bootstrap.min.js"></script>
    </head>
    <body>
        <div class="container">
            <div class="span10 offset2">
                <div class="row">
                    <h3>Read a Person</h3>
                </div>
                <form class="form-horizontal" method="post">
                    <div class="control-group">
                        <label class="control-label">Name</label>
                        <div class="controls">
                            <input name="name" type="text" placeholder="Name" disabled="disabled" value=<?php echo(empty($data['Name']) ? $data['Name'] : '') ?>">
                        </div>
                    </div>
                    <div class="control-group">
                        <label class="control-label">Surname</label>
                        <div class="controls">
                            <input name="surname" type="text" placeholder="Surname" disabled="disabled" value=<?php echo(empty($data['Surname']) ? $data['Surname'] : '') ?>">
                        </div>
                    </div>
                    <div class="control-group">
                        <label class="control-label">City</label>
                        <div class="controls">
                            <input name="city" type="text" placeholder="City" disabled="disabled" value=<?php echo(empty($data['City']) ? $data['City'] : '') ?>">
                        </div>
                    </div>
                    <div class="control-group">
                        <label class="control-label">Birthdate</label>
                        <div class="controls">
                            <input name="birthdate" type="text" placeholder="Birthdate (yyyy-mm-dd)" disabled="disabled" value=<?php echo(empty($data['Birthdate']) ? $data['Birthdate'] : '') ?>">
                        </div>
                    </div>
                    <div class="form-actions">
                        <a class="btn" href="index.php">Back</a>
                    </div>
                </form>
            </div>
        </div>
    </body>
</html>

```

Figura 2.135: read.php

- *update.php*:

```

<?php
    require 'database.php';
    $id = null;
    if ( !empty($_GET['id'])) {
        $id = $_REQUEST['id'];
    }

    if ( null==$id ) {
        header("Location: index.php");
    }
    if ( !empty($_POST)) {
        // keep track validation errors
        $nameError = null;
        $surnameError = null;
        $cityError = null;
        $birthdateError = null;

        // keep track post values
        $name = $_POST['name'];
        $surname = $_POST['surname'];
        $city = $_POST['city'];
        $birthdate = $_POST['birthdate'];

        // validate input
        $valid = true;
        if (empty($name)) {
            $nameError = 'Please enter Name';
            $valid = false;
        }
        if (empty($surname)) {
            $surnameError = 'Please enter Surname';
            $valid = false;
        }
        if (empty($city)) {
            $cityError = 'Please enter City';
            $valid = false;
        }

        // update data
        if ($valid) {
            $pdo = Database::connect();
            $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            $sql = "UPDATE person set name = ?, surname = ?, city = ?, birthdate = ? WHERE id = ?";
            $q = $pdo->prepare($sql);
            $q->execute(array($name, $surname, $city, $birthdate, $id));
            Database::disconnect();
            header("Location: index.php");
        }
    } else {
        $pdo = Database::connect();
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        $sql = "SELECT * FROM person where id = ?";
        $q = $pdo->prepare($sql);
        $q->execute(array($id));
        $data = $q->fetch(PDO::FETCH_ASSOC);
        $name = $data['Name'];
        $surname = $data['Surname'];
        $city = $data['City'];
        $birthdate = $data['Birthdate'];
        Database::disconnect();
    }
?>

```

Figura 2.136: update.php

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <script src="js/bootstrap.min.js"></script>
</head>
<body>
    <div class="container">
        <div class="span10 offset1">
            <div class="row">
                <h3>Update a Person</h3>
            </div>

            <form class="form-horizontal" action="update.php?id=<?php echo $id ?>" method="post">
                <div class="control-group <?php echo!empty($nameError) ? 'error' : '' ?>">
                    <label class="control-label">Name</label>
                    <div class="controls">
                        <input name="name" type="text" placeholder="Name" value="<?php echo!empty($name) ? $name : '' ?>">
                        <?php if (!empty($nameError)): ?>
                            <span class="help-inline"><?php echo $nameError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($surnameError) ? 'error' : '' ?>">
                    <label class="control-label">Surname</label>
                    <div class="controls">
                        <input name="surname" type="text" placeholder="Surname" value="<?php echo!empty($surname) ? $surname : '' ?>">
                        <?php if (!empty($surnameError)): ?>
                            <span class="help-inline"><?php echo $surnameError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($cityError) ? 'error' : '' ?>">
                    <label class="control-label">City</label>
                    <div class="controls">
                        <input name="city" type="text" placeholder="City" value="<?php echo!empty($city) ? $city : '' ?>">
                        <?php if (!empty($cityError)): ?>
                            <span class="help-inline"><?php echo $cityError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="control-group <?php echo!empty($birthdateError) ? 'error' : '' ?>">
                    <label class="control-label">Birthdate</label>
                    <div class="controls">
                        <input name="birthdate" type="text" placeholder="Birthdate" value="<?php echo!empty($birthdate) ? $birthdate : '' ?>">
                        <?php if (!empty($birthdateError)): ?>
                            <span class="help-inline"><?php echo $birthdateError; ?></span>
                        <?php endif; ?>
                    </div>
                </div>
                <div class="form-actions">
                    <button type="submit" class="btn btn-success">Update</button>
                    <a class="btn" href="index.php">Back</a>
                </div>
            </form>
        </div>
    </div>
</body>
</html>

```

Figura 2.137: update.php

- *delete.php*:

```

<?php
require 'database.php';
$id = 0;

if (!empty($_GET['id'])) {
    $id = $_REQUEST['id'];
}

if (!empty($_POST)) {
    // keep track post values
    $id = $_POST['id'];

    // delete data
    $pdo = Database::connect();
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $sql = "DELETE FROM person WHERE id = ?";
    $q = $pdo->prepare($sql);
    $q->execute(array($id));
    Database::disconnect();
    header("Location: index.php");
}
?>

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link href="css/bootstrap.min.css" rel="stylesheet">
        <script src="js/bootstrap.min.js"></script>
    </head>

    <body>
        <div class="container">

            <div class="span10 offset1">
                <div class="row">
                    <h3>Delete a Person</h3>
                </div>

                <form class="form-horizontal" action="delete.php" method="post">
                    <input type="hidden" name="id" value="<?php echo $id; ?>" />
                    <p class="alert alert-error">Are you sure to delete ?</p>
                    <div class="form-actions">
                        <button type="submit" class="btn btn-danger">Yes</button>
                        <a class="btn" href="index.php">No</a>
                    </div>
                </form>
            </div>
        </div>
    </body>
</html>

```

Figura 2.138: delete.php

Navighiamo ora nell'applicazione:

The screenshot shows two browser windows. The top window is titled "My PHP CRUD" and contains a table with columns: Name, Surname, City, Birthdate, and Action. A red arrow points from the bottom of this window down to the second window. The second window is titled "Create a new Person" and has input fields for Name, Surname, City, and Birthdate, along with "Create" and "Back" buttons.

Figura 2.139: MyCRUD - Create new Person

This screenshot displays four browser windows illustrating the creation of a new person. The first window shows validation errors: "Name" and "Surname" are required, and "City" is missing. The second window shows the same form with filled fields: Name (Mario), Surname (Rossi), City (Milano), and Birthdate (1975-11-03). The third window shows the successful creation of the record in the database. The fourth window shows the updated list of people in the "My PHP CRUD" table, with the newly created record (Mario Rossi, Milano, 1975-11-03) listed.

Figura 2.140: MyCRUD - Create new Person 2

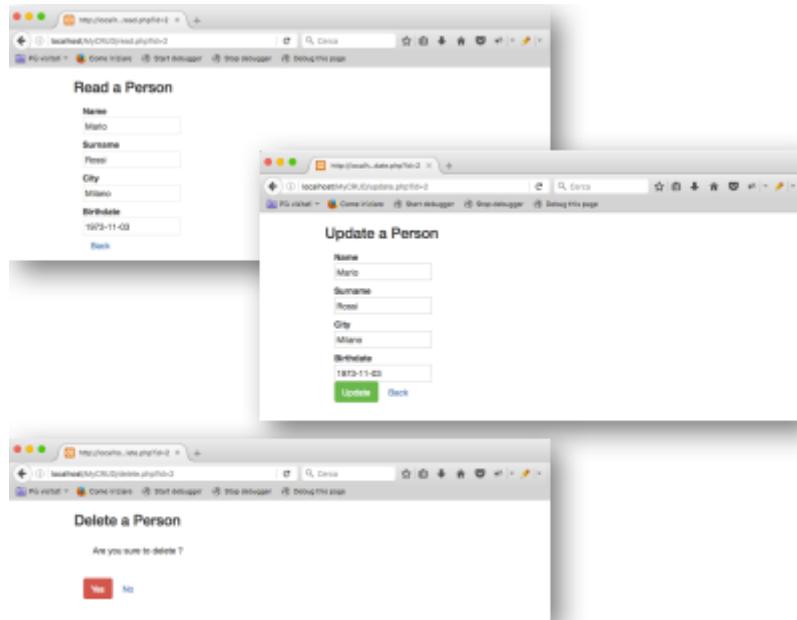


Figura 2.141: MyPHP CRUD - Read, Update and Delete a Person

Lucia Vaira
03/11/2016

2.11 Enhanced Mapping

Quando ci si addentra più a fondo nei dettagli del modello concettuale, sfruttando il paradigma EER anziché il classico ER, si ha bisogno di conoscere una versione estesa del noto Algoritmo di Mapping che permette il passaggio dal Modello Concettuale al Modello Logico. Esistono diverse tipologie di DB, quelli da cui siamo partiti sono i database relazionali ma ne esistono di diversi tipi. I **database Key-Value** sono una naturale generalizzazione dei costrutti Key-Value implementati da vari linguaggi di programmazione, come ad esempio il PHP, che utilizza lo stesso paradigma per implementare i suoi array associativi. Questi meccanismi di storage sono caratterizzati dall'indicizzazione degli elementi tramite una chiave; e viceversa, dato un oggetto campione, si può risalire alle chiavi corrispondenti. Questa particolare tecnica di indicizzazione è ancora in realtà molto vicina al concetto di JOIN tradizionale. Per i **database Document-based** ogni oggetto non è rappresentato come una semplice tupla, ma è visto come un documento, proprio nel senso informatico del termine (documento XML o JSON ad esempio). Questi database hanno caratteristiche comuni con i database Object-Oriented, un'altra categoria di DB basata sulle astrazioni del paradigma OOP, soltanto che per questa classe di database ogni colonna è in realtà un oggetto più complicato, ed a differenza dei DB-OO non ci sono metodi. Troviamo anche i **database colonnari**, anche detti schemaless DB, per i quali i dati sono organizzati per colonne anziché per righe o tuple come nei relazionali. Eventuali proprietà o attributi possono essere aggiunti dinamicamente. Infine, abbiamo i **database a grafi**, ove i principali elementi costitutivi sono nodi ed archi, e ragionevolmente le relazioni possono essere intese come archi che collegano nodi. Questi ultimi database sono utili per modellare i dati di molti problemi scientifici, oppure alcune interazioni all'interno delle reti nei Social Network. Esistono quindi differenti tipi di database, e per ognuno abbiamo differenti strumenti

per rappresentare il modello concettuale (UML, ER, etc.). È pertanto naturale considerare una versione estesa e completa dell'algoritmo di mapping.

2.11.1 ALGORITMO DI MAPPING

Fondamentalmente, come abbiamo già visto, il mapping è una sequenza di regole che consentono di passare dal modello concettuale al modello logico. I tipi di entità diventano sempre tabelle, per i tipi di relazione invece bisogna considerare altri fattori (tipicamente la loro molteplicità). La versione completa dell'algoritmo di mapping si compone di nove differenti passi, che prendono in esame anche situazioni più specifiche e particolari, come ad esempio relazioni deboli, di possesso, attributi multipli etc.

- **Regola 1:** Tutti gli Entity Type diventano delle tabelle. Per ogni entità costruiamo quindi una tabella. Questo è generalmente vero se abbiamo solo degli attributi semplici all'interno del tipo di entità;
- **Regola 2:** Trasformare i tipi di entità deboli (Weak Entity Type) in tabelle. Dobbiamo creare una nuova tabella, la cui chiave primaria sarà costituita dalla coppia Primary Key dell'Owner e Partial Key della entità debole. Eventualmente dovremmo metterci anche degli attributi aggiuntivi riguardanti la relazione;
- **Regola 3, Relazione binaria: {1:1}:** Ci sono tre possibili soluzioni:
 - (External Key Approach): Includere la relazione dal lato destro o dal lato sinistro;
 - (Merged Relationship Approach): Accoppiare le due tabelle formando un'unica tabella;
 - (Cross Reference o Relationship Relation Approach): Anche se la relazione è 1:1, potremo comunque voler decidere di avere una tabella separata per la relazione, bisogna ovviamente tenere conto delle entità in gioco, e soprattutto tenere sempre a mente che bisogna minimizzare il numero di NULL. Potrebbe essere conveniente, ad esempio, unificare le due tabelle in un'unica tabella quando abbiamo una PARTECIPAZIONE TOTALE sia a destra che a sinistra.

In generale il fatto che si possa semplificare, non è detto che alla fine implichì che lo si debba fare: bisogna sempre rispettare dei principi base, come la minimizzazione del numero di NULL appena richiamata;

- **Regola 4, Relazione binaria: {1:N}:** In questo caso, si avranno in generale 2 tabelle e si include nella tabella sul lato N dell'associazione la chiave primaria della entity type che contribuisce con molteplicità 1 all'associazione. Ma ancora una volta, possiamo in maniera lecita decidere di mantenere separate le due tabelle ed aggiungerne un'altra per la relazione. Non dobbiamo dimenticare che, qualora optassimo per la prima scelta infatti, comunque gli attributi della relazione verrebbero inclusi nel lato N dell'associazione, e qualora ci possano essere essere molti NULL, forse potrebbe esser meglio considerare il secondo approccio;
- **Regola 5, Relazione binaria: {M:N}:** Avremo 3 tabelle, 2 relative alle entità che partecipano alla relazione e la terza relativa alla relazione stessa. Nella tabella relativa alla relazione, saranno presenti gli attributi semplici della relazione;

- **Regola 6, Composed and MultiValued Attributes:** Nel caso di attributi composti, dobbiamo dividerli in attributi semplici. Nel caso di attributo MultiValued, dobbiamo trasformare questo attributo in una tabella separata e modellarlo come un Weak Entity Type, pertanto la chiave primaria della nuova tabella sarà la coppia della chiave primaria della entity type owner e della chiave parziale. Chiaramente la relazione che si verrà a creare sarà 1:N;
- **Regola 7, Relazioni N-arie:** Qui dobbiamo prestare particolare attenzione: potremmo avere una "semplice" relazione ternaria, ma anche delle relazioni che coinvolgono, per l'appunto N entità. In tal caso questa relazione diviene sempre una nuova tabella che include TUTTE le chiavi primarie delle entità coinvolte nella relazione. Queste PK assieme formano la chiave primaria della tabella.

Naturalmente se la relazione ha degli attributi semplici sono inclusi sempre nella nuova tabella. Un'alternativa alle relazioni N-arie è la reificazione, come già sappiamo. In tal caso la relazione diviene un tipo di entità e quindi mappata in tabella, che è collegata alle N entità in gioco, rispettivamente con N relazioni 1:N. A questo punto possiamo capire quale è uno dei problemi del mapping dal mondo relazionale al mondo Object Oriented. Sostanzialmente, quando creiamo delle relazioni, abbiamo bisogno degli identificativi delle entità in gioco. Porre questi identificativi, prendendo in esame, ad esempio, una relazione 1:N in una classe che rappresenta un tipo di entità, costituirebbe una violazione del principio di INCAPSULAMENTO, che a questo punto capiamo non essere contemplato dal modello ER. Non solo, ma alle volte in una relazione M:N per la quale abbiamo bisogno anche di attributi aggiuntivi, bisognerebbe creare un'altra classe separata soltanto per modellare la relazione, violando nuovamente l'incapsulamento. D'altro canto, è lo stesso paradigma OOP che ha presentato dei limiti nel corso del tempo, tanto è vero che si è tentati di colmare le sue lacune introducendo nuovi concetti come il Contract Oriented Programming, Subject Oriented Programming, oppure paradigmi più complessi che trattano i Soft Objects o Clubject come oggetti fondamentali, potendo essere trattati come classi ed entità allo stesso tempo;

- **Regola 8, Oggetti:**

- **Regola 8A, Superclassi e sottoclassi (specializzazioni):** Abbiamo diverse possibilità. La prima è mantenere la superclasse e le sottoclassi in tabelle separate. Si deve avere l'accortezza di avere la stessa chiave primaria della superclasse nelle sottoclassi;
- **Regola 8B:** Possiamo spostare tutti gli attributi della superclasse nelle sottoclassi. In questo modo avremo soltanto le tabelle relative alle classi specializzate, ovvero alle sottoclassi. La PK della superclasse viene preservata ma si introduce ridondanza e potrebbe insorgere il problema della presenza di NULL nelle sottoclassi;
OSSERVAZIONE: è importante osservare che gli attributi che costituiscono una PK concettuale non sono MAI generati automaticamente.
Ci sono quindi differenti opzioni. Qual è la scelta giusta? In questo caso, le differenti opzioni da vagliare sono anche più complesse. Bisogna sempre salvaguardare il rispetto dei principi di base, uno dei quali consiste nella minimizzazione del numero di NULL;
- **Regola 8C:** Si possono eventualmente eliminare sottoclassi e spostare i relativi attributi nella superclasse. In questo caso si aggiunge un nuovo attributo TIPO nella superclasse (esempio: tipo → ingegnere o tecnico);

- **Regola 8D:** Come la regola 8C ma si inseriscono più attributi TIPO; (esempio: tipo_mestiere → ingegnere, tipo_specializzazione → civile)

Con l'opzione 8A dobbiamo mantenere le tabelle della superclasse e delle sottoclassi. Le query richiederanno obbligatoriamente l'utilizzo delle JOIN, e quindi risulteranno più dispendiose. Mentre negli altri casi, sebbene disponiamo subito del tipo di specializzazione, abbiamo un eventuale problema di sparsità di attributi, dovuto alla presenza di possibili NULL. Sta a noi scegliere il giusto compromesso. È importante osservare che nell'opzione 8B si dovranno eventualmente duplicare le relazioni che prima erano collegate con la superclasse.

Ereditarietà Multipla: concetto assolutamente legittimo che si verifica quando dobbiamo ereditare da più superclassi. Ci possono addirittura essere delle situazioni ove abbiamo due percorsi distinti per arrivare allo stesso genitore, in tal caso si erediterà due o più volte dallo stesso genitore. Questi casi non sono gestiti a livello relazionale e a livello OOP. La gestione di questi casi è dipendente dal contesto e va eventualmente gestita nel livello applicativo;

- **Regola 9, Mapping delle CATEGORIE (o UNIONI):** Abbiamo già detto che non possiamo implementare le generalizzazioni, ma possiamo utilizzare un loro surrogato. Questo è il concetto di UNION. Ad esempio possiamo avere il concetto di proprietario di un determinato prodotto. Tale proprietario può essere una Banca, un'Impresa od una singola Persona. Notiamo che queste entità non hanno attributi comuni però possiamo ugualmente assimilarli ad un concetto comune di proprietario utilizzando una Surrogate Key. Avremo una chiave Owner.Id che sarà presente come ulteriore colonna per ogni tipo di entità coinvolto nella unione, e sarà utilizzata come chiave esterna per referenziare queste entità in una apposita relazione.

2.12 Overview

Overview sul programma svolto finora:

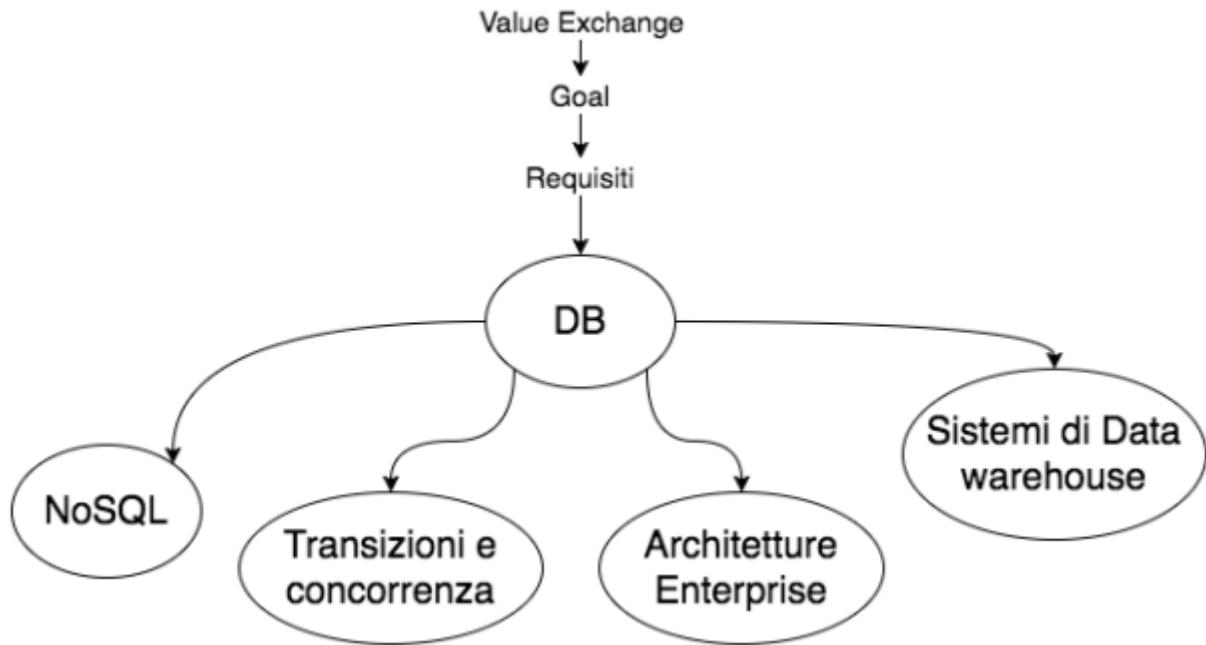


Figura 2.142: Overview generale

Parlando in ottica generale, noi siamo partiti dal concetto di Database. I nostri studi che abbiamo fatto e faremo si basano su quattro possibili macro-aspetti:

- **Tipo di database:** come abbiamo detto ci sono differenti tipi di database, ad esempio i relazionali ed i NoSQL. È da citare il CAP-THEOREM, che riguarda, nell'ambito dei DB NoSQL il trade-off che sussiste tra COERENZA e SCALABILITÀ. Non si possono avere entrambi gli aspetti contemporaneamente, e diversi tipi di database preferiscono favorire l'una o l'altra proprietà. I database non relazionali ad esempio, possono accettare una perdita di coerenza, ma al contempo consentono un'elevata scalabilità. Tra i DB NoSQL si possono citare ad esempio MongoDB e Spark;
- **Transazioni e Concorrenza:** in particolare è importante ricordare il set di proprietà ACID;
- **Architettura Enterprise:** nelle applicazioni in cui è necessario dover gestire una rete di computer interconnessi, si parla di System of Systems;
- **Sistemi multidimensionali, oppure Data Warehouses:** che riguardano come suggerisce il nome stesso, dei veri e propri database di database.

2.12.1 Modellazione di dati

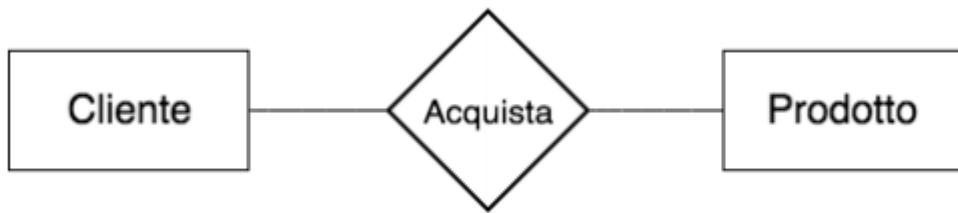


Figura 2.143: Cliente Acquista Prodotto

Il più semplice database Cliente-Acquista-Prodotto, nella sua semplicità, riveste una elevatissima importanza. Guardando lo schema del genere ci si può accorgere di che particolare modello di sistema, di business, di presentazione abbiamo bisogno. Per quanto riguarda il modello di sistema è importante considerare il carico di lavoro che il nostro sistema dovrà sopportare, dimensionalmente espresso in [query/s]. Sulla base del carico di lavoro e dell'ordine di grandezza della quantità di dati in gioco bisogna tener in conto di aspetti come la ridondanza dei dati o di sistemi di backup incrementali o differenziali.

Un altro aspetto molto importante da tenere in conto è la struttura che viene ancor prima della modellazione dei nostri DB. Esiste un sistema sovrastante che si compone essenzialmente di tre blocchi principali: VE = Value Exchange, G = Goals, R = Requirements. I Requisiti nell'ingegneria del SW sono una rappresentazione dei desideri dei clienti in determinato momento ma questi possono cambiare nel tempo. I GOAL sono invece gli obiettivi da raggiungere, le motivazioni, gli obiettivi ultimi che stanno dietro all'espressione di un desiderio da parte di un cliente. I GOAL sono robusti, ovvero invarianti rispetto al contesto. Finché un'azienda rimane un'azienda di vendite, il GOAL primario sarà sempre quello di vendere!

I GOAL esprimono un problema, laddove i REQUISITI esprimono una specifica soluzione ad una specifica istanza del problema!

Si può identificare uno spazio di problemi definiti dai GOAL. Per ogni problema esistono molteplici soluzioni, ognuna delle quali è basata sull'utilizzo di una specifica tecnologia. Progettare in fin dei conti, significa capire quali sono i goal e qual è il problema. Per capire bene l'interazione che avviene tra Goal e Requisiti bisogna seguire due regole fondamentali, tenendo sempre presente la libertà strategica del committente, ovviamente. Queste regole o modus operandi sono:

- **Raffinamento:**

Abbiamo un certo numero di stakeholders, ognuno dei quali ha un determinato goal da soddisfare, alcuni dei quali eventualmente condivisi tra più stakeholders differenti. Per raffinamento i goal vengono via via espressi in sottogoal, fino ad arrivare ad una rappresentazione granulare dei goal dalla quale l'estrazione dei requisiti è molto semplice. I requisiti rappresentano una possibile risoluzione ad uno o più goal. Si parte quindi da un concetto astratto e si arriva ad un concetto concreto. A tal proposito è utile la costruzione di due diagrammi:

- **Diagramma Goal-Stakeholders:** ove abbiamo la rappresentazione di tutti gli stakeholders e di tutti i goal ad essi relativi e per ogni stakeholder tracciamo degli archi orientati verso i goal che intendono soddisfare/raggiungere. Esiste una sottile analogia con i diagrammi UseCases UML:

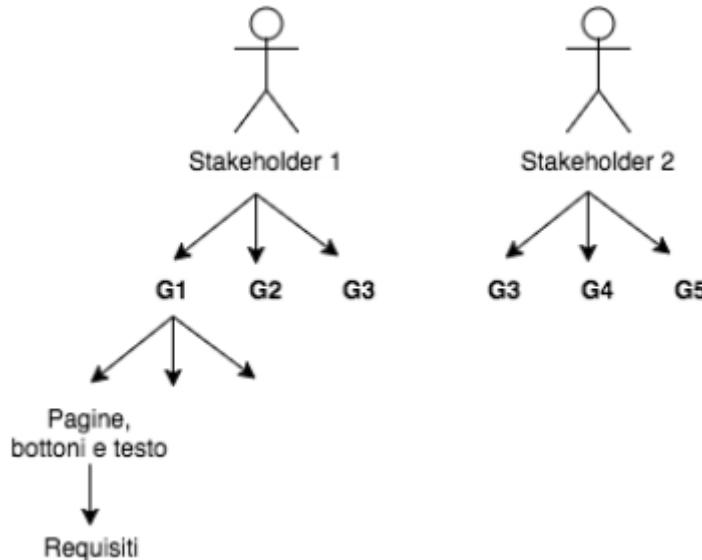


Figura 2.144: Diagramma Goal-Stakeholders

- Il raffinamento può essere eseguito utilizzando una struttura tabellare per la quale come attributi troviamo: stakeholder, goal (numerati, in modo tale da poter indizzarli di nuovo con lo stesso identificativo in caso di overlapping) e subgoal che da essi si dipartono. Allo stesso modo si dovrà poi creare una tabella dimensionalmente uguale ma che tratta invece semanticamente i requisiti come soluzione ad ogni goal espresso. Quando copriamo tutti i GOAL dobbiamo poter cominciare l'implementazione. I requisiti rappresentano quindi le foglie di questa struttura ad albero generica prima descritta:

Stakeholder	Goal	Sub-Goal
S1 - Negoziante	G1 - Vendere online G2 - Controllo acquisti e merce G3 - Controllo frodi	Pagamento Paypal Pagamento CC
S2 - Cliente
S3 - Addetto Vendite

Figura 2.145: Tabella Goal-Stakeholders

- **Diagramma dello scambio di valore (E3Value):**

Abbiamo citato prima il VE = Value Exchange. È sostanzialmente un modello di business. Significa letteralmente scambio di valore, e rappresenta per l'appunto le entità ed i flussi di valore che intercorrono tra esse. Un diagramma rappresentante uno scambio di valore può considerarsi completo quando tutti i cicli di scambio di valore vengono chiusi. Vedasi

il modello di Google ad esempio. Apparentemente è un "innocuo" motore di ricerca, ma un'analisi dettagliata di business porta alla luce come in realtà esso rappresenti un ecosistema molto esteso ed intricato, dal punto di vista dello scambio di valore. Gli utenti prendono da Google delle informazioni effettuando delle ricerche, e Google prende a sua volta i dati di queste richieste, ottenendo dei PROFILI degli utenti. I siti web pagano Google per avere un rank più alto. Aderendo al programma AdSense inoltre, i siti Web ricevono del denaro per ogni visualizzazione dei banner pubblicitari ospitati sui siti aderenti. Questa pubblicità è opportunamente targettizzata, ed in ultima analisi va a beneficio di alcune Business Partner di Google, che pagano quest'ultimo per esporre la loro pubblicità. Esiste inoltre un'altra serie di Business Partner che pagano Google per ottenere i profili che esso ha raccolto nelle ricerche degli utenti e li utilizzano per effettuare delle ricerche di mercato.

Di seguito è riportato il diagramma E3V per il caso di studio di Google.

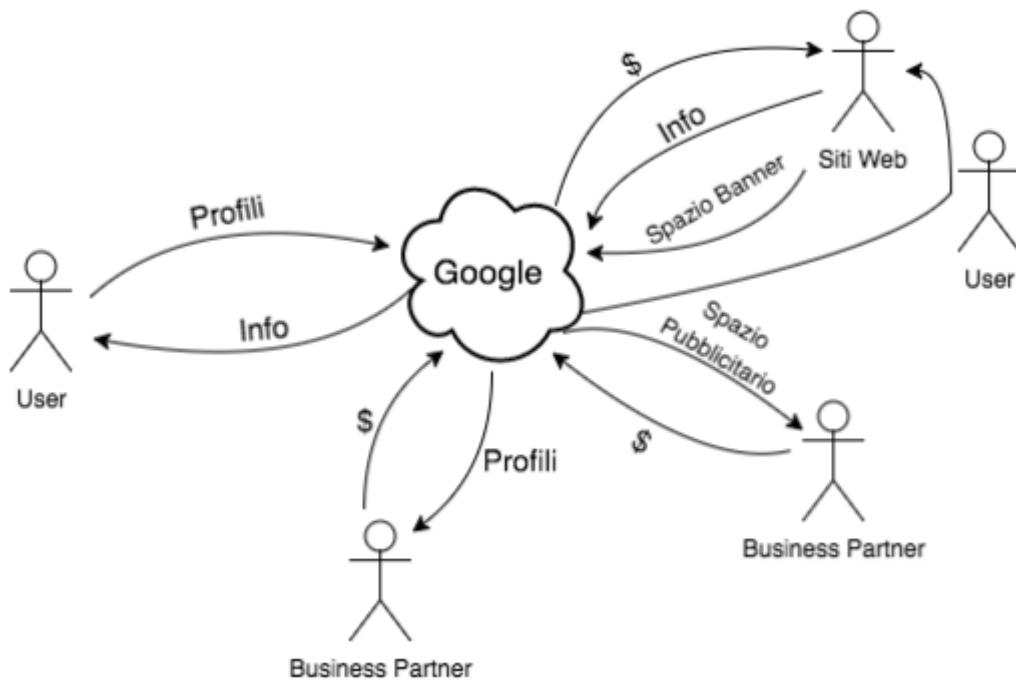


Figura 2.146: Diagramma E3VALUE

Gabriele Accarino
Marco Chiarelli
09/11/2016

2.13 Mapping RECAP

Ricapitoliamo l'ALGORITMO DI MAPPING:

- Ogni entità regolare si trasforma in una tabella, che include tutti gli attributi dell'entità;
- Gli attributi composti vengono scomposti in attributi semplici;

- Gli attributi multi-valore diventano una tabella, in cui c'è un attributo corrispondente all'attributo multi-valore e la chiave primaria dell'entità (come chiave esterna):



Figura 2.147: ESEMPIO

- Ogni entità debole diventa una tabella in cui è inclusa la chiave primaria dell'owner come chiave esterna:



Figura 2.148: ESEMPIO: TRENO possiede VAGONE

- Con una relazione m:n si hanno 3 tavelli: una per ciascuna entità e una per la relazione, che contiene le chiavi primarie delle entità come chiavi esterne;
- Con una relazione 1:n si hanno 2 tavelli, una per ciascuna entità. La tabella dell'entità sul lato n contiene anche gli attributi della relazione e la chiave primaria dell'altra entità come chiave esterna;
- Con una relazione ricorsiva si hanno due alternative:
 - Nel caso di relazione 1:n si ha un'unica tabella con una chiave esterna che punta alla chiave primaria:

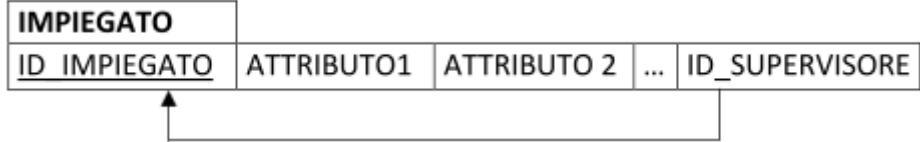


Figura 2.149: ESEMPIO: IMPIEGATO supervisiona IMPIEGATO

- Nel caso di relazione m:n si ha una tabella per l'entità e una per la relazione, in cui le due chiavi esterne puntano alla chiave primaria dell'entità:

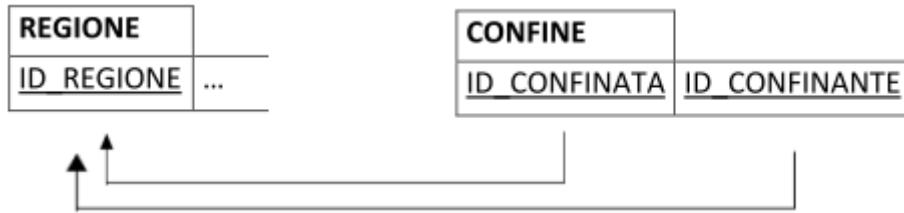


Figura 2.150: ESEMPIO: REGIONE confina con REGIONE

- Con una relazione 1:1, si hanno 3 possibilità:
 - **Approccio basato su chiavi esterne:** si include la relazione nell'entità in cui è presente la partecipazione totale (per ridurre il numero di NULL)



Figura 2.151: ESEMPIO: IMPIEGATO dirige DIPARTIMENTO

- **Approccio basato su relazione fusione:** quando si ha partecipazione totale da entrambi i lati si può scegliere di:
 - * Unire le entità in un'unica tabella, contenente gli attributi di entrambe le entità e della relazione:



Figura 2.152: ESEMPIO: PRESIDENTE presiede STATO

- * Comportarsi come con una relazione m:n (3 tabelle distinte);
- **Approccio basato su relazione associazione:** ci si comporta come per una relazione m:n (3 tabelle distinte).
- Con una relazione ternaria si hanno 4 tabelle, una per ogni entità e una per la relazione.

2.13.1 Svolgimento traccia d'esame

Corso di Database – Computer Engineering Esercitazione 10 Novembre 2016

Un'officina meccanica, concessionaria per la manutenzione di una famosa casa automobilistica, vuole automatizzare la gestione della sua clientela, del relativo parco auto e degli interventi in officina. Oltre alla manutenzione ordinaria e straordinaria delle auto della casa automobilistica, inviategli dai concessionari di zona, essa possiede un parco clienti con modelli di auto appartenenti a svariate case automobilistiche. La concessionaria intende realizzare un sistema che le permetta di automatizzare i seguenti aspetti:

- L'anagrafica dei clienti, distinti in privati (nome, cognome, codice fiscale, indirizzo, contatti – email, telefono, cellulare, etc.) e aziende (ragione sociale, PIVA, indirizzo, contatti – email, telefono, cellulare, etc.), delle automobili (modello, targa, data di immatricolazione, data di acquisto, etc.), dei concessionari di auto con cui ha contatti e relazioni commerciali, del personale addetto agli interventi (nome, cognome, codice fiscale, qualifica – meccanico, elettrauto, gommista, etc.) e della merce in magazzino (codice, descrizione, prezzo di acquisto, prezzo di vendita, data di acquisto, fornitore, etc.);
- La prenotazione degli interventi di manutenzione ordinaria e straordinaria e il ritiro dell'auto all'indirizzo indicato dal cliente. Il sistema suggerirà la data in base alla disponibilità del personale e del magazzino (es. olio, pezzi di ricambio, etc.), riportando il tempo medio di durata dell'intervento e la data prevista di ritiro. Per interventi di durata superiore alle 4 ore il cliente può richiedere un'auto di cortesia tra quelle disponibili nel periodo indicato, che gli dovrà essere consegnata presso un indirizzo concordato;
- La gestione degli interventi in officina, della merce in magazzino e del personale che ha effettuato l'intervento;
- La notifica delle manutenzioni ordinarie ai clienti che lo richiedono. Tale notifica può avvenire via cellulare o via email.

Per lo scenario descritto:

1. Si definisca un diagramma ER
2. Si derivi il corrispondente modello relazionale
3. Si implementino in linguaggio SQL le seguenti interrogazioni:
 - a. Elenco degli interventi di manutenzione e del personale che li ha eseguite sull'auto targata "AB123YB";
 - b. Elenco degli interventi effettuati nel 2008 su auto FIAT per il cambio delle gomme;
 - c. Totale fatturato effettuato dalla società GHIS nel primo trimestre dell'anno 2009.

Figura 2.153: Esercitazione Database - 10 Novembre 2016

Cominciamo con l'analisi della traccia:

Un'officina meccanica, concessionaria per la manutenzione di una famosa casa automobilistica, vuole automatizzare la gestione della sua clientela, del relativo parco auto e degli interventi in officina. Oltre alla manutenzione ordinaria e straordinaria delle auto della casa automobilistica, inviategli dai concessionari di zona, essa possiede un parco clienti con modelli di auto appartenenti a svariate case automobilistiche. La concessionaria intende realizzare un sistema che le permetta di

Abbiamo un'officina che vuole gestire:

- CLIENTELA;
- AUTOMOBILI;
- INTERVENTI

Possiamo iniziare ad individuare le prime entità che vogliamo modellare:

svariate case automobilistiche. La concessionaria intende realizzare un sistema che le permetta di automatizzare i seguenti aspetti:

- L'anagrafica dei clienti, distinti in privati (nome, cognome, codice fiscale, indirizzo, contatti – email, telefono, cellulare, etc.) e aziende (ragione sociale, P.IVA, indirizzo, contatti – email, telefono, cellulare, etc.), delle automobili (modello, targa, data di immatricolazione, data di acquisto, etc.), dei concessionari di auto con cui ha contatti e relazioni commerciali, del personale addetto agli interventi (nome, cognome, codice fiscale, qualifica – meccanico, elettrauto, gommista, etc.) e della merce in magazzino (codice, descrizione, prezzo di acquisto, prezzo di vendita, data di acquisto, fornitore, etc.);

Per gestire l'anagrafica dei clienti, distinti in privati e aziende, usiamo una specializzazione dell'entità **CLIENTE**. La specializzazione è disgiunta perché un cliente è un privato oppure un'azienda, non può essere entrambi. Notiamo che ci sono attributi comuni tra privati e aziende, come *indirizzo* e *contatti*, che sono entrambi attributi composti.

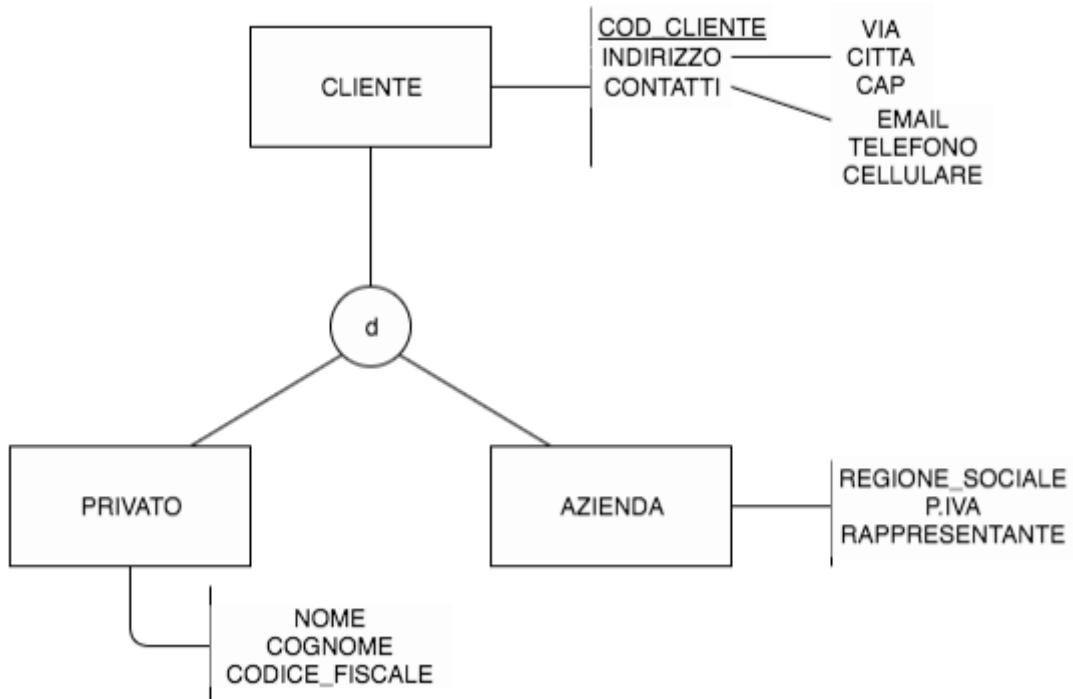


Figura 2.154: Specializzazione entità CLIENTE

Creiamo l'entità **AUTO**. Verrebbe spontaneo usare la targa come chiave primaria, ma si tratta di una stringa, quindi è più comodo a livello di prestazioni usare un id auto incrementale, più facile da gestire. L'attributo *colore* è un attributo multiplo.



Figura 2.155: Entità AUTO

L'attributo *cortesia* modella il concetto che ci sono automobili del concessionario che possono essere usate dai clienti come auto di cortesia.

Creiamo l'entità **CONCESSIONARIO**:

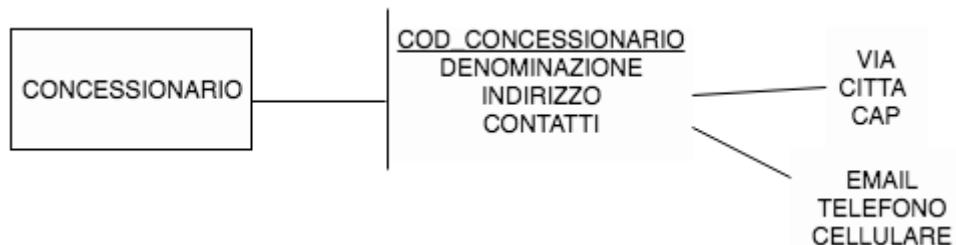


Figura 2.156: Entità CONCESSIONARIO

Creiamo l'entità **DIPENDENTE**:

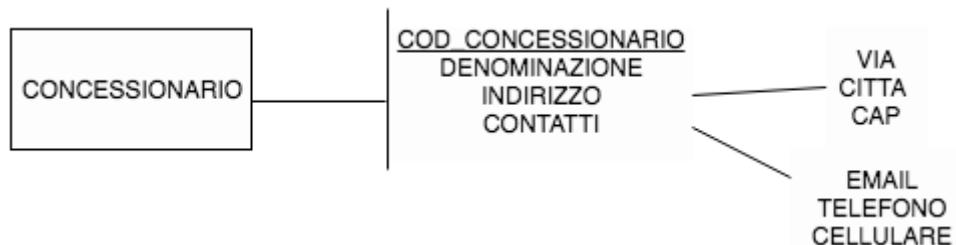


Figura 2.157: Entità DIPENDENTE

Per modellare la merce in magazzino creiamo l'entità **PRODOTTO**:

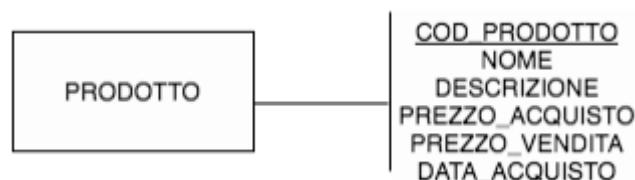


Figura 2.158: Entità PRODOTTO

Un prodotto è fornito da un **FORNITORE**, che è meglio modellare come entità a sé:

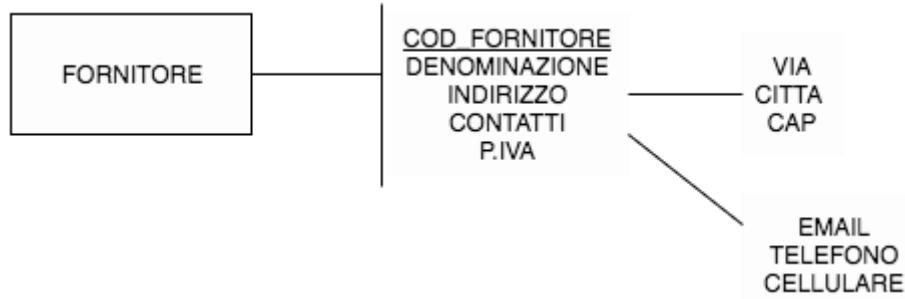


Figura 2.159: Entità FORNITORE

- La prenotazione degli interventi di manutenzione ordinaria e straordinaria e il ritiro dell'auto all'indirizzo indicato dal cliente. Il sistema suggerirà la data in base alla disponibilità del personale e del magazzino (es. olio, pezzi di ricambio, etc.), riportando il tempo medio di durata dell'intervento e la data prevista di ritiro. Per interventi di durata superiore alle 4 ore il cliente può richiedere un'auto di cortesia tra quelle disponibili nel periodo indicato, che gli dovrà essere consegnata presso un indirizzo concordato;
- La gestione degli interventi in officina, della merce in magazzino e del personale che ha effettuato l'intervento;
- La notifica delle manutenzioni ordinarie ai clienti che lo richiedono. Tale notifica può avvenire via cellulare o via email.

Creiamo l'entità **TIPO INTERVENTO**, che specializziamo:

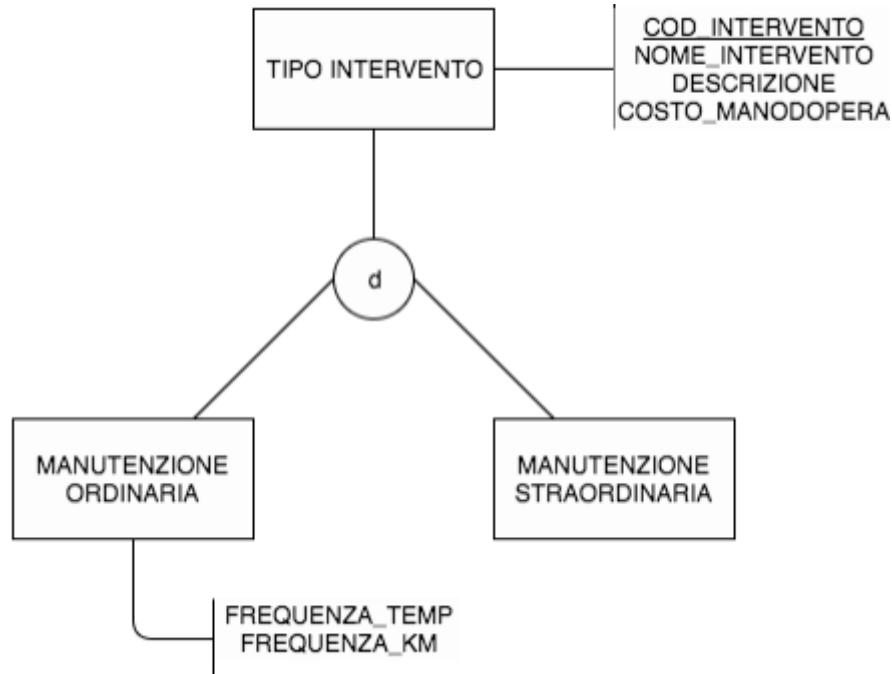


Figura 2.160: Entità TIPO INTERVENTO

Occupiamoci ora di modellare le relazioni.

Un cliente **POSSIEDE** un'auto oppure **RICHIEDE** un'auto come vettura di cortesia:

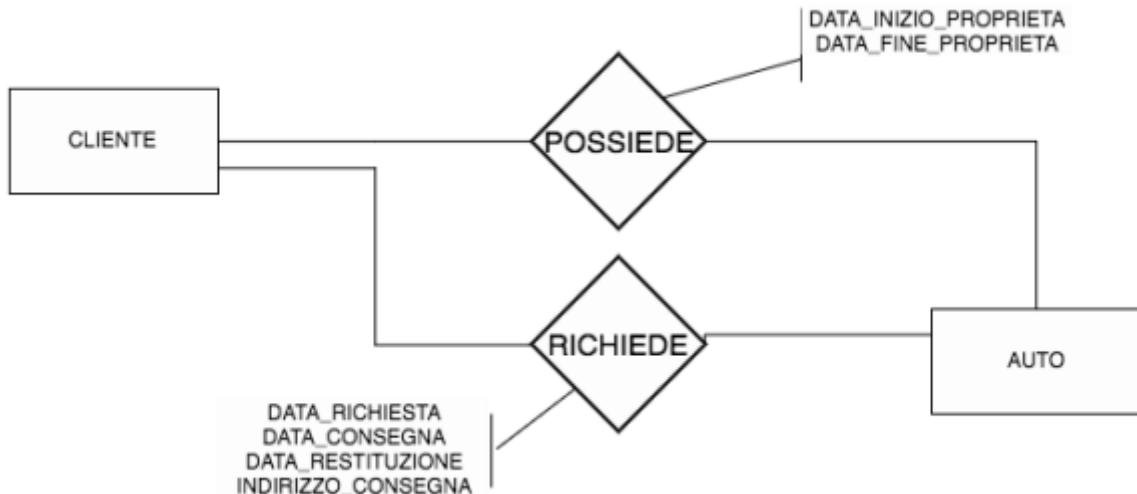


Figura 2.161: Relazione cliente POSSIEDE/RICHIEDE auto

L'auto può anche appartenere al concessionario:



Figura 2.162: Auto APPARTIENE a concessionario

Per tenere traccia del fatto che un cliente richiede un intervento per un'auto creiamo la relazione:

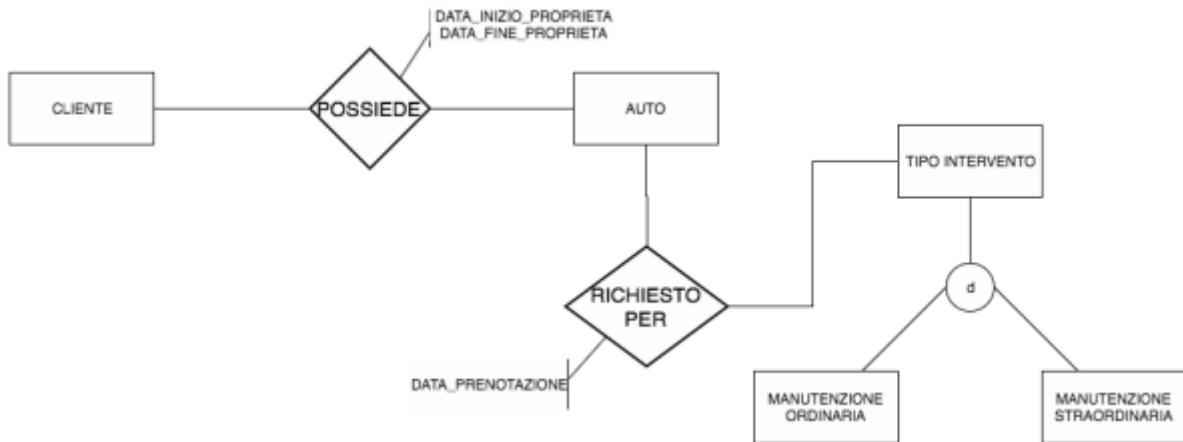


Figura 2.163: Tipo Intervento RICHIESTO PER auto

L'auto subisce un intervento (che può essere un insieme di diversi tipi di interventi). Distinguiamo tipo intervento dall'intervento specifico creando l'entità **INTERVENTO**:

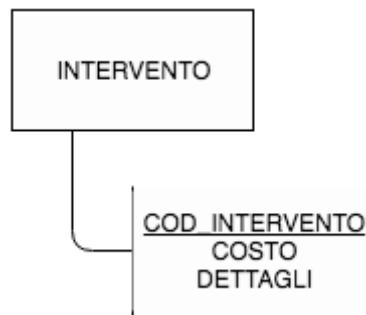


Figura 2.164: Entità INTERVENTO

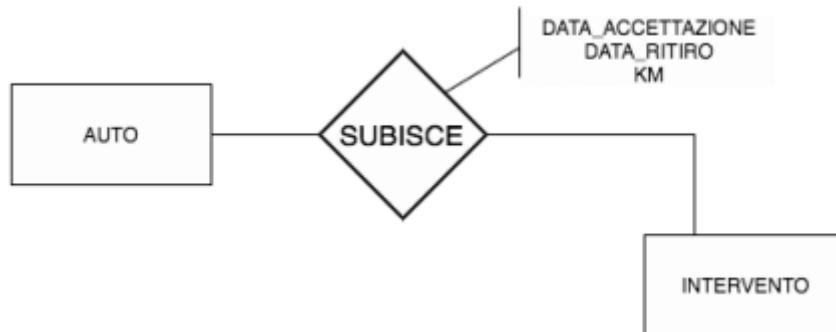


Figura 2.165: Relazione Auto SUBISCE Intervento

Un intervento è costituito da diversi tipi di interventi, ciascuno eseguito da un dipendente (relazione ternaria):

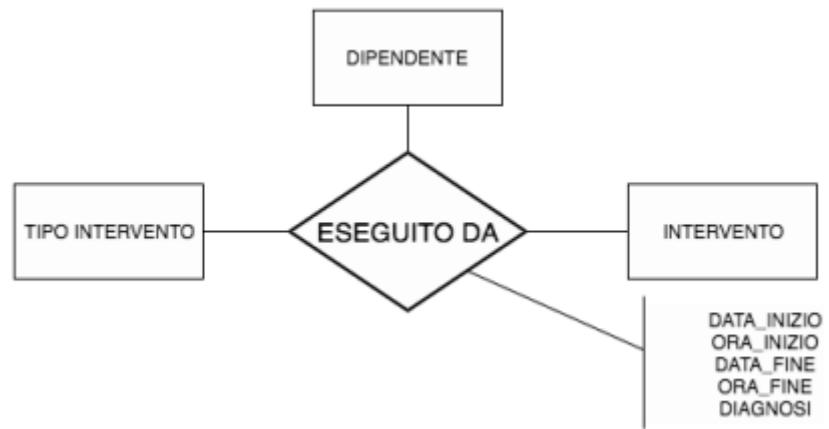


Figura 2.166: Relazione ternaria ESEGUITO DA

Per un intervento sono usati dei prodotti, acquistati da un fornitore:



Figura 2.167: Relazione Prodotto ACQUISTATO DA Fornitore

Per tenere traccia del fatto che un cliente richiede una notifica per la manutenzione ordinaria su un'auto creiamo la relazione:

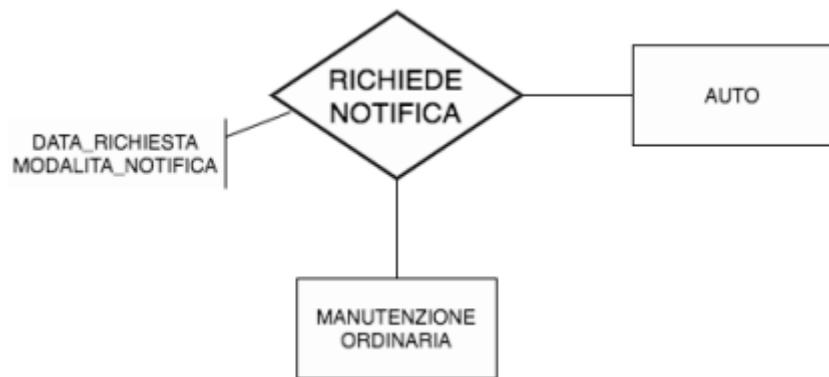


Figura 2.168: Relazione Auto RICHIEDA NOTIFICA per Manutenzione Ordinaria

Il diagramma EER finale è:

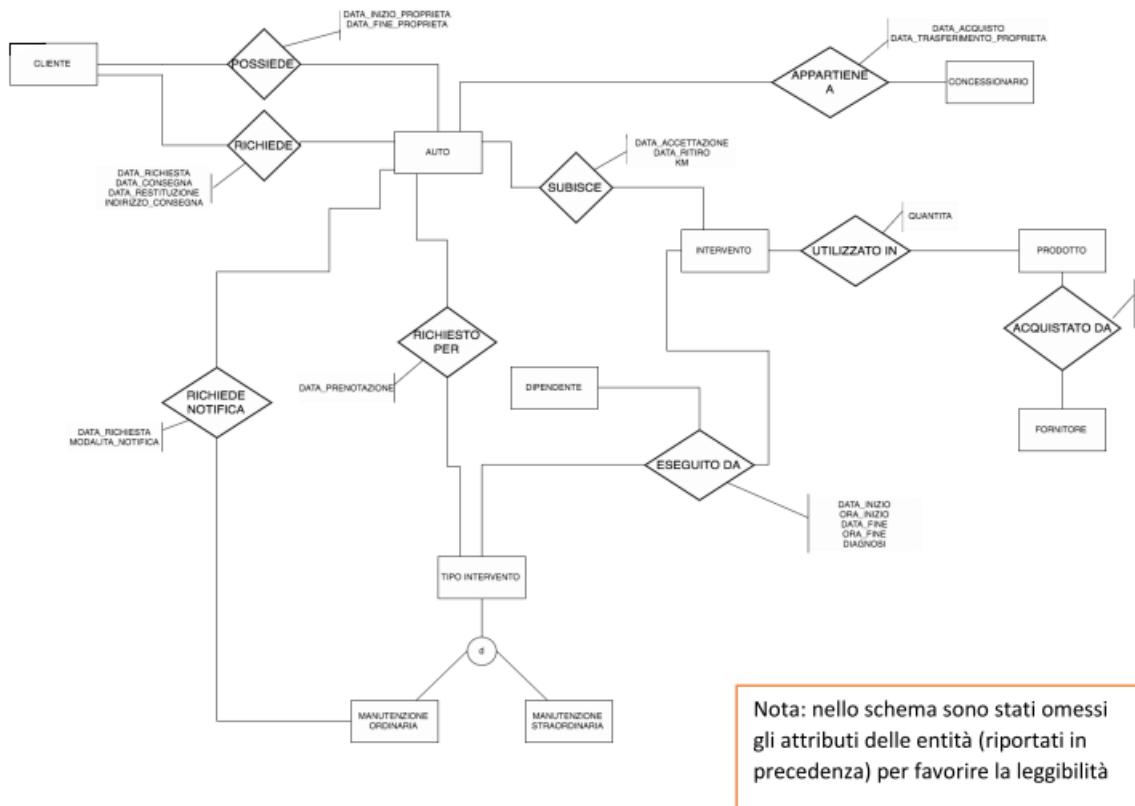


Figura 2.169: Schema ER finale per l'Esercitazione 28/11/2016

Floriana Accoto
Marco Mameli
10/11/2016

2.14 Qualità di un DB

Lo scopo della lezione è quello di comprendere quando un DB è di qualità. Per fare ciò è necessario comprendere il concetto di dipendenze funzionali, normalizzazione e denormalizzazione.

La valutazione di un software/database può essere:

- Soggettiva (Ad esempio il software/DB è comodo da usare per l'utente che lo ha implementato o per un generico utente. In questo caso non si possono fare misure sulla qualità);
- Oggettiva (Se il database deve essere di qualità, bisogna normalizzarlo, pertanto si ricorre al concetto di dipendenza funzionale, la quale è una misura di qualità oggettiva di progettazione, di cui si parlerà più in dettaglio in seguito).

Le 3 regole fondamentali, viste fino ad ora, per una buona progettazione di un database sono:

- Minimizzare la presenza di attributi a cui viene assegnato il valore NULL;
- Minimizzare la ridondanza di informazione;
- Evitare, a meno di casi eccezionali, la cancellazione dei dati.

Esiste un modo per eliminare del tutto la ridondanza all'interno di un database.

Ad esempio partendo dall'entità Persona, si potrebbe pensare di realizzare una nuova relazione Persona possiede Nome, per evitare che uno stesso nome si ripeta più volte all'interno del database. Lo stesso discorso si può fare con l'entità nome, a sua volta composta da lettere che si ripetono, quindi si crea una nuova relazione del tipo Nome è composta da Lettera. Si può iterare questo procedimento fino ad avere un database composto da un'entità di bit, che assumerà come valori solo 0 e 1, e una lunga serie di relazioni, dato che tutta la conoscenza che abbiamo si può esprimere sotto forma di permutazioni di bit.

E' questo il modo di ridurre del tutto la ridondanza all'interno di un database, ma non è una strategia perseguitabile in quanto la mente umana non ragiona in bit e per di più si dovrebbe fare una serie di join molto lunga. Pertanto è preferibile lavorare con i tipi di dati, l'SQL e non con i bit (anche se poi in realtà il funzionamento della macchina, che all'occhio umano è nascosto, è proprio di quel tipo descritto sopra).

Ad esempio, oggi si parla spesso di database universali, cioè, detto in maniera informale, di database che non hanno bisogno di fare delle join per recuperare i dati, ma di semplici Select. Ciò è dovuto al fatto che si ha a che fare con un'unica tabella che al suo interno presenta molta ridondanza di informazioni. E' proprio quello che accade con i BIG DATA, che sono dei schemaless DB, cioè le tabelle di questo tipo di DB vengono riempite con qualsiasi tipo di informazione memorizzabile.

Il concetto di BIG DATA è caratterizzato dalla regola delle 3 V:

- Velocity (Si vogliono trattare delle informazioni in tempo reale e non a posteriori. Esempio: il problema dello streaming);
- Variety (Non può esistere un modello informativo univoco in tutto il web, ognuno ha le sue regole. Si ha a che fare con una quantità di informazioni diverse tra loro);
- Volume (Si vuole avere a disposizione quante più informazioni possibili o lavorare con delle informazioni così vaste a tal punto dal poter considerare di avere tutta l'informazione possibile).

Uno dei problemi legati ai BIG DATA è proprio legato al fatto che si hanno delle tabelle universali. Potenzialmente potrei avere righe e colonne infinite, dato che la ridondanza è molto elevata. Google utilizza dei DB di questo genere per offrire il proprio servizio. Nel seguente schema è rappresentata, in maniera indicativa, la relazione tra il numero di join da effettuare in base alla ridondanza di dati all'interno del DB e anche il numero di join da effettuare in base alla quantità di informazioni che si hanno.

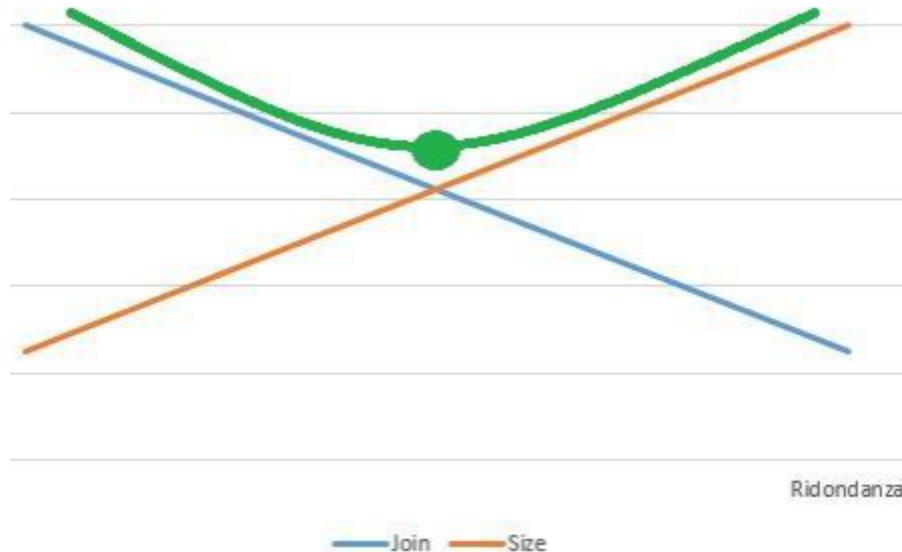


Figura 2.170: Diagramma Trade/Off $|Join|/Size$

In corrispondenza di bassa ridondanza, il numero di join da effettuare aumenta e viceversa (retta blu). Allo stesso modo, all'aumentare della dimensione dei dati rappresentati in bit, aumentano il numero di join e viceversa (retta arancione). Quello che bisogna fare è trovare un giusto compromesso. Facendo così il prodotto tra le due rette, cioè tra la quantità di dati da gestire e la quantità di calcolo da effettuare, si ottiene una parabola che rappresenta la maneggevolezza del DB, il cui minimo rappresenta un DB di qualità. Naturalmente, in base alle esigenze del committente, non sempre si possono fare delle scelte che portano ad una soluzione di qualità (DB giusto). La velocità di calcolo dipende dalla quantità di memoria che voglio impiegare. La quantità di memoria (lo spazio) e la quantità di calcolo (il tempo) sono informazioni duali, se una diminuisce tende ad aumentare l'altra. Per trovare la soluzione ottima devo avere una quantità ragionevole sia di memoria che di calcolo.

Per progettare un DB di qualità è necessario seguire delle linee guida. Gli step fondamentali da seguire sono 4:

- Un DB deve essere facile da spiegare. La semantica utilizzata deve essere chiara e semplice. Chiariamo il concetto di entità/relazione → sono rappresentati da un insieme di attributi che hanno un ciclo di vita comune.

Es. Gli attributi NOME, COGNOME, INDIRIZZO... associati ad un'entità studente hanno un ciclo di vita comune, cioè sono noti tutti allo stesso tempo in fase di inserimento. Mentre ESAME non è da considerare come attributo di uno studente poiché non si può sapere a priori se uno studente sosterrà un dato esame e quale sarà la sua votazione. Quello delle UPDATE ANOMALIES è un altro problema di cui tener conto. Devo fare in modo che nel mio DB non appaiano o scompaiano informazioni in maniera indesiderata. Quindi è sbagliato considerare come attributi alcuni valori che attraverso degli update potrebbero sparire dal DB. Es. Creare una tabella impiegato_dipartimento potrebbe portare a delle UPDATE ANOMALIES. Le anomalie si verificano in fase di:

- INSERT (si vuole aggiungere un impiegato, ma si è costretti a lasciare vuote le informazioni sul dipartimento);

- DELETE (per poter cancellare un dipartimento, si deve eseguire una cancellazione in verticale in modo che non si tocchino gli impiegati);
- UPDATE (anche se si possono eseguire update parziali, si avranno anomalie nel momento in cui il dipartimento resta senza impiegati sparando dal database).
- Bisogna disegnare lo schema relazionale senza anomalie, che a volte devono esserci necessariamente a causa delle richieste del committente. In tal caso sarà necessario gestire queste anomalie a livello di codice (Business Rule). Es. Si deve gestire a livello software il fatto che alcune persone non hanno un codice fiscale che permette loro di essere identificate univocamente;
- Limitare/eliminare i valori di tipo NULL. Anche in questo caso, ci potrebbe essere la possibilità di non poterli eliminare, allora si dovrà giustificare dettagliatamente la loro presenza nel DB;
- Disegnare schemi relazionali senza tuple spurie, cioè con delle join che non soddisfano nessun vincolo di chiave esterna. Es. Se si fa una NATURAL JOIN tra due tabelle che hanno due colonne con lo stesso nome, queste verranno messe in join e il risultato conterrà dei valori errati.

2.14.1 DIPENDENZE FUNZIONALI

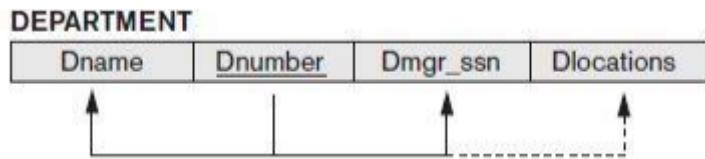


Figura 2.171: Entità DIPARTIMENTO

La dipendenza funzionale non è altro che una relazione tra un gruppo di attributi e altri attributi. In riferimento alla figura sopra si ha che il Dnumber è in relazione con Dname, Dmgr_ssn, Dlocation. Un po' come la PK, la quale essendo identificatore unico di una tupla, permette di risalire facilmente agli altri attributi ad essa associata. Es. SSN (PK) implica la conoscenza di nome, cognome, data di nascita... E' anche vero però che se si conoscono nome, cognome, data di nascita... si può risalire all'SSN.

Sul concetto di dipendenze funzionali si basano i vari tipi di forme normali di un DB.

- **PRIMA FORMA NORMALE:**

Un DB relazionale è in prima forma normale se non ha attributi multipli e attributi composti (gli attributi devono essere degli scalari e non dei vettori);

- **SECONDA FORMA NORMALE:**

Un DB relazionale è in seconda forma normale se tutte le dipendenze funzionali sono totali. Le dipendenze funzionali possono essere anche parziali, cioè un gruppo di attributi può dipendere da un attributo e un altro gruppo può dipendere da un altro attributo diverso dal precedente. In tal caso è necessario “spezzare” tutte le dipendenze funzionali parziali in modo da ottenere solo dipendenze funzionali totali.

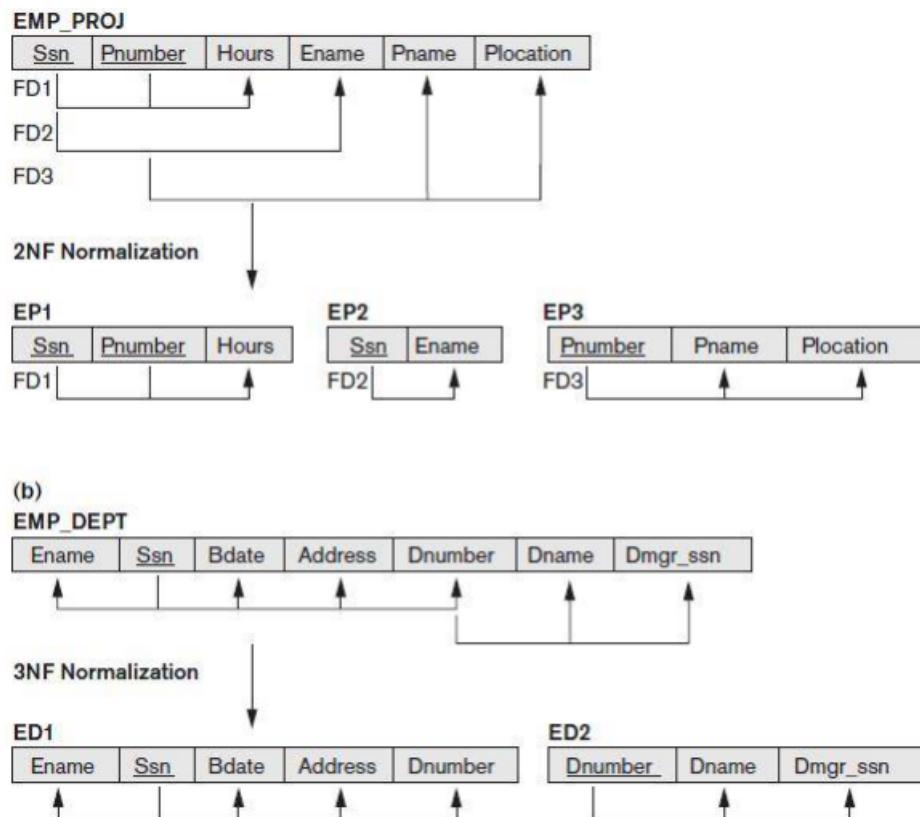


Figura 2.172: Seconda FORMA NORMALE

- **TERZA FORMA NORMALE:**

Un DB relazionale è in terza forma normale se non ci sono dipendenze funzionali transitive. Si parla di dipendenze funzionali transitive se dall'attributo A dipende un gruppo di attributi B e da uno degli attributi B dipende un gruppo di attributi C. Anche in questo caso di risolve “spezzando” la dipendenza funzionale transitiva in più dipendenze funzionali:

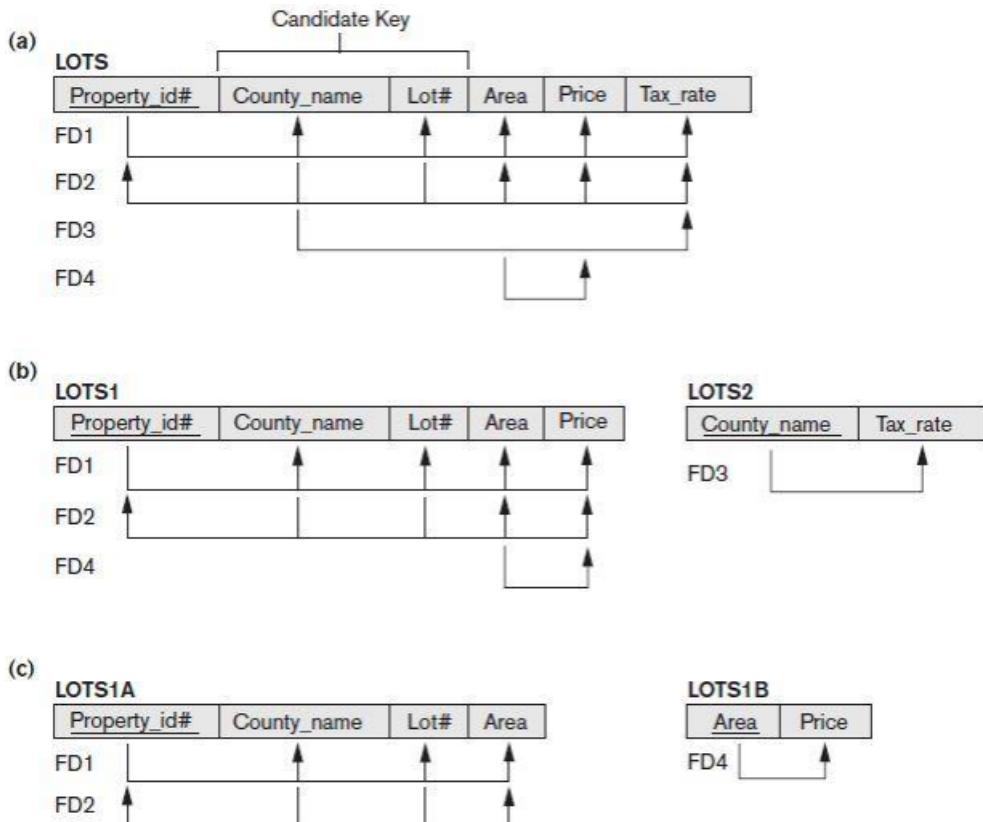


Figura 2.173: Terza FORMA NORMALE

- **TERZA FORMA NORMALE DI BOYCE-CODD:**

Un DB relazionale è in terza forma normale di BOYCE E CODD se non ci sono dipendenze funzionali ricorsive del tipo $A \rightarrow B$ e $B \rightarrow A$. In questo caso di genera un loop che è difficile da spezzare, perché significherebbe perdere informazione. Si risolve, senza perdere informazioni, quando si implementa il codice, facendo dei controlli a livello di Business.

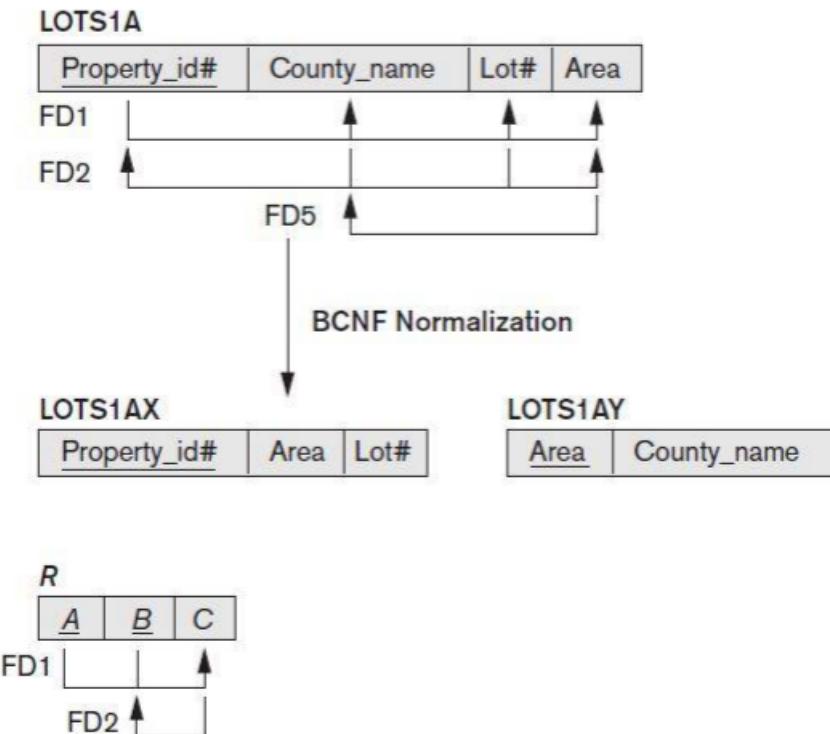


Figura 2.174: Terza FORMA NORMALE di BOYCE-CODD

Mediante relazione matematica si scrive che:

3a FORMA NORMALE → 2a FORMA NORMALE → 1a FORMA NORMALE

Cristian Annicchiarico
Mattia Marzano
10/11/2016

2.15 Transazioni e Concorrenza

2.15.1 Transazioni

Introduzione al Processing Transazionale

Uno dei criteri per classificare un Database è in base al numero di utenti che vi accedono in maniera concorrente. Si distingue tra:

- Single-User System: un solo utente utilizza il sistema (non attuale);
- Multi-User System: più processi in esecuzione in parallelo, concorrentemente;
- Concorrenza: le risorse condivise sono le tabelle. Questa situazione si divide a sua volta in:
 - Interleaved Processing: il processore riserva diversi time-slots per ogni task attraverso un lavoro di scheduling (multitasking);

- Parallel Processing: più processori lavorano in parallelo con un utilizzo oculato della cache.

Le operazioni CRUD sono atomiche ma, in determinate situazioni come ad esempio il trasferimento di somme di denaro, sono necessarie più operazioni indivisibili.

A questo punto si introduce il concetto di *Transazione*: unità logica di database processing che include una o più operazioni di accesso (read-retrieval, write-insert or update, delete). Un fondamentale aspetto delle transazioni consiste nell'avere un unico risultato, Success or Failure, e di conseguenza vengono viste come un unico oggetto.

I quattro oggetti fondamentali di un DBMS sono:

- Tabelle;
- Viste;
- Stored Routine:
 - Stored Procedure;
 - Stored Function.

Le Stored Procedure sono un modo semplice, elegante e robusto da un punto di vista matematico per affrontare le transazioni. Proprio queste rappresentano il punto di forza dei Database relazionali.

Two Sample Transactions

Osserviamo ora due transazioni di esempio:

(a)	T_1	(b)	T_2
	<pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>		<pre>read_item(X); X := X + M; write_item(X);</pre>

Figura 2.175: Due transazioni di esempio

In questo caso il problema consiste nella mancanza di garanzia riguardo l'ordine delle operazioni. L'obiettivo è garantire il principio di conservazione che, nel caso di bonifico bancario o di un generico pagamento implica che non debbano esserci perdite di denaro.

Quando queste due transazioni sono eseguite si possono incontrare diversi tipi di problemi:

- Lost Update Problem:

T_1	T_2
read_item(X); $X := X - N;$	
write_item(X); read_item(Y);	read_item(X); $X := X + M;$
$Y := Y + N;$ write_item(Y);	write_item(X);

Time ↓

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Figura 2.176: Lost Update Problem

In questo caso il valore finale di X è incorretto in quanto è stato perso il write di T_1 per un errato lavoro di scheduling da parte del processore;

- Temporary Update (or Dirty Read) Problem:

T_1	T_2
read_item(X); $X := X - N;$ write_item(X);	
read_item(Y);	read_item(X); $X := X + M;$ write_item(X);

Time ↓

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the temporary incorrect value of X .

Figura 2.177: Temporary Update Problem

In generale, questa situazione può verificarsi se T_1 fallisce: ad esempio mentre il processore esegue le istruzioni di T_2 il conto di Y può non esistere più e la read successiva può restare bloccata; quindi la transazione T_1 va annullata, ma nel frattempo T_2 ha lavorato basandosi su dati sbagliati. In questo caso non è possibile tornare allo stato iniziale in quanto T_2 non conosce lo stato iniziale di T_1 .

- Incorrect Summary Problem:

T_1	T_2
<pre> read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y); </pre>	<pre> sum := 0; read_item(A); sum := sum + A; ⋮ read_item(X); sum := sum + X; read_item(Y); sum := sum + Y; </pre> <p style="text-align: right;">← T_2 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).</p>

Figura 2.178: Incorrect Summary Problem

Se una transazione sta calcolando una funzione somma aggregata su dei dati mentre un'altra li sta aggiornando, la funzione aggregata può calcolare alcuni valori prima del loro aggiornamento e dopo che altri hanno subito l'update.

State Transition Diagram

Lo State Transition Diagram illustra gli stati per l'esecuzione di una generica transazione:

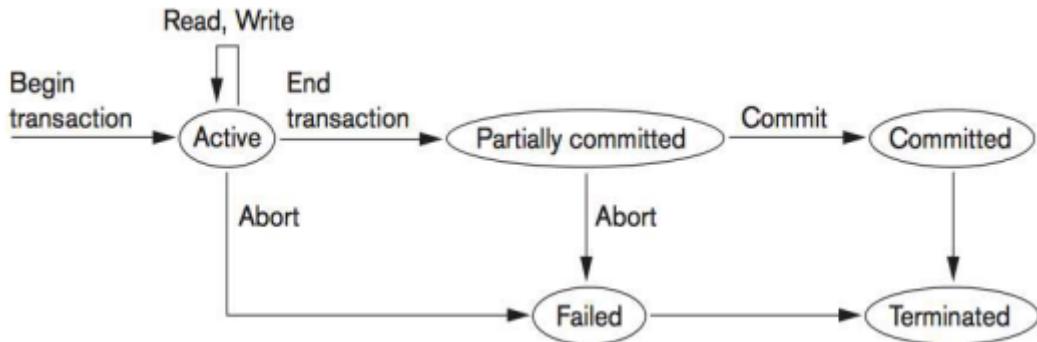


Figura 2.179: State Transition Diagram

In questa situazione, viene preferita la semplicità del pensiero sequenziale all'efficienza del calcolo parallelo, immaginando che questo non esista.

Si ponga particolare attenzione allo stato **Partially committed**, dove, attraverso un processo automatizzato verificato internamente, viene verificato che non ci siano situazioni errate nel database. Se le operazioni sono corrette viene infine effettuato il **Commit** (SAVE).

Nel caso in cui ci si trovasse nello stato **Failed**, invece, viene effettuato un Rollback in modo tale da non cambiare lo stato del database evitando i salvataggi (UNDO).

Questo approccio presenta due problemi principali: impegna molto il processore a causa del non utilizzo del parallelismo (tale onerosità è dovuta alla crescita più che lineare in termini computazionali) e non permette al Database relazionale di essere scalabile. Tuttavia comporta un notevole vantaggio: tale approccio risulta robusto e semplice grazie all'approccio matematico sottostante e l'utilizzo del codice sequenziale.

Proprietà Transazione

- Atomicità: la transazione deve essere indivisibile;
- Consistenza: il database passa da uno stato valido ad un altro stato valido;
- Isolamento: ogni transazione agisce come se sia isolata dalle altre;
- Durabilità (o permanenza).

Le transazioni che rispettano tali principi sono dette ACID (dalle *in* proprietà).

Isolation Level	Type of Violation		
	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Figura 2.180: Tipi di Transazione

Riguardo alla tabella precedente, si noti come i livelli di acidità crescano verso il basso mentre le performance migliorino verso l'alto. Anche se più veloci, i primi tre Isolation Level non devono mai essere usati in scrittura per write precise.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

Figura 2.181: Transazione di esempio

2.15.2 Concurrency Control Techniques

Two Phases Locking Techniques

Nei Database viene usato un lock particolare avente due modalità:

- Shared (read);
- Exclusive (write).

Più persone possono accedere in lettura ad una risorsa senza generare conflitti (shared lock): il lock è come se non esistesse. La fase più delicata è quella di scrittura, durante la quale chi sta eseguendo il write deve notificare la propria azione a chi accede in lettura a tale dato.

Segue la Conflict Matrix relativa alla situazione appena descritta:

	Read	Write
Read	Yes	No
Write	No	No

Figura 2.182: Conflict Matrix

Il protocollo di locking a due fasi funziona in questo modo: inizialmente blocca tutto, successivamente utilizza le risorse e infine sblocca tutto. In tal caso il parallelismo dei processori viene sfruttato, e la coda gestita automaticamente grazie ai lock:

- Lock → Growing (accresciamo il lock set);
- Unlocking → Shrinking.

In tal caso il vantaggio principale risulta essere la velocità di esecuzione, in quanto vengono impiegate meno risorse; lo svantaggio invece l'elevata complessità.

Figure 21.4
Transactions T_1' and T_2' , which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol.
Note that they can produce a deadlock.

T_1'	T_2'
<code>read_lock(Y); read_item(Y); write_lock(X); unlock(Y) read_item(X); $X := X + Y;$ write_item(X); unlock(X);</code>	<code>read_lock(X); read_item(X); write_lock(Y); unlock(X) read_item(Y); $Y := X + Y;$ write_item(Y); unlock(Y);</code>

Figura 2.183: Vari tipi di lock

Deadlock Detection e Resolution

Un Deadlock rappresenta un loop dal quale non si esce: la transazione A aspetta B e viceversa. Un deadlock dinamico, o starvation, mette un timeout alla transazione: dopo tale intervallo di tempo di inattività, la transazione viene bloccata e riparte automaticamente. Il problema consiste nella possibilità che la transazione sia semplicemente troppo lunga, quindi rispetto al deadlock standard la macchina lavora all'infinito.

Locking ottimistico e pessimistico

Il locking ottimistico lavora sui dati ipotizzando che non ci siano altri oggetti che utilizzano le sue risorse. Successivamente vengono effettuate delle verifiche che, se valide, permettono di effettuare la write. In caso contrario si verifica se chi accede ai dati contesi lo fa in scrittura o in lettura, eventualmente effettuando un rollback. In sintesi il funzionamento è il seguente:

- Read Phase;
- Validation Phase;
- Write Phase.

2.16 Indicizzazione e Indici

Un albero rappresenta una partizione su un dataset in cui i nodi vengono suddivisi finché non si arriva alle “foglie”, ovvero gli elementi di minore granularità. Nel caso dei Database gli alberi rappresentano tutti i livelli di aggregazione su tutti i possibili livelli.

Le tabelle occupano un certo spazio in memoria in termini di byte: per ottimizzare le prestazioni vengono utilizzate tecniche di pooling, in cui si associano dei blocchi di id per ogni macchina che lavora sul database. In realtà gli id sono dei numeri pseudo casuali piuttosto che sequenziali, in quanto più onerosi; tale approccio è giustificato dal fatto che, nei database reali, la probabilità di trovare due id uguali è molto bassa. In tal caso la situazione viene gestita nella maniera opportuna rieseguendo le operazioni: ciò che è importante è che nonostante tale eventualità questa situazione risulta essere la più efficiente.

Si definisce quindi un indice per ogni criterio, indicizzando la tabella in modo che possa essere ordinata secondo tutti i criteri possibili. Automaticamente, tramite una linked list trasparente a chi gestisce i database, ad ogni nuovo inserimento viene ordinata una tabella aggiuntiva. Viene quindi fatta una join (complessità $O(n)$).

Posizionando il dato al posto giusto ad ogni inserimento, creando opportunamente nuovi id e shiftando i valori successivi, la complessità scende da $O(n \log n)$ a $O(n)$, grazie alla struttura dati di tipo indice. Viene quindi spostata la ricerca da runtime a inserttime.

Con le indicizzazioni vengono create delle strutture invisibili a noi, automaticamente; ciò che viene deciso consiste in quali colonne debbano essere indicizzate. Si noti che la chiave primaria di una tabella è sempre indicizzata in quanto necessaria per le operazioni di join.

Tutto ciò comporta un grande vantaggio: ogni ricerca di una colonna indicizzata costa poca memoria in più sul disco, comportando un notevole guadagno di tempo. Si tratta, ovviamente, di un trade-off più che conveniente nella maggior parte delle applicazioni.

Lorenzo Caputo
Ippazio Alessio
16/11/2016

Capitolo 3

DATA WAREHOUSE

3.1 Introduzione

Finora abbiamo parlato dei database relazionali. I data warehouse non sono database fatti per il CRUD, ma sono adatti soprattutto per il calcolo o l'analisi dati (letteralmente data warehouse significa magazzino di dati) e differiscono dai database perché possono essere visti come una sorta di database di database.

Esistono dei prodotti software per costruire data warehouse; nel mondo open source abbiamo:

- Pentaho

Invece nel mondo del business abbiamo delle estensioni che permettono di gestire il multidimensionale, quali:

- Oracle;
- Analisys Services (un'estensione di Microsoft SQL Server).

Per certi versi il mondo dei data warehouse somiglia al mondo dei database relazionali: anche qui useremo la modellazione a tre livelli (concettuale, logica e fisica), tuttavia il mondo dei database analitici è diverso per altre ragioni.

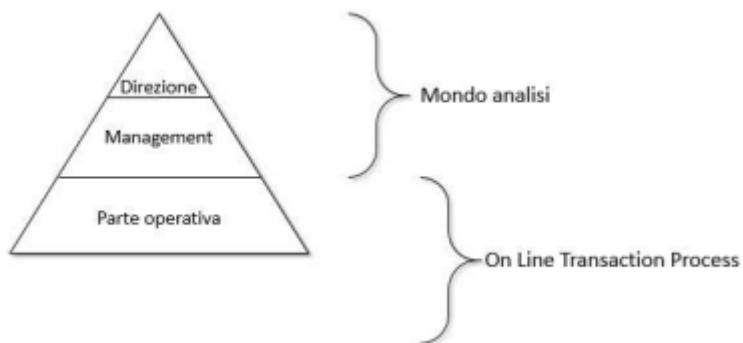


Figura 3.1: Piramide DMO di Anthony

Alla base della piramide di Anthony vi è il livello del transazionale (On Line Transaction Processing) che può essere visto come un'operazione CRUD o un insieme di relazioni CRUD interpretate come un unico oggetto. Questo è il mondo delle transazioni, diverso da quello dell'analisi. Cominceremo a prendere dimestichezza con l'ambito dei data warehouse quando avremo a che fare non tanto con gli aspetti operativi delle aziende, quanto piuttosto con quelli manageriali o dirigenziali. Questi progetti sono molto costosi: a differenza di quelli transazionali che vanno da mille a un milione di euro, quelli di data warehouse hanno un costo da alcune decine di milioni di euro in su. Un progetto può arrivare a costare tanto poiché richiede lo sforzo di creare il contesto lavorativo giusto nel quale far funzionare l'informatica.

In genere nella pubblica amministrazione si dispone delle informazioni, si aggregano, sintetizzano e le si inviano ai piani superiori. Questo insieme di operazioni hanno dei "Key Performance Indicator" e sono dei dati sintetici rivolti ai dirigenti. Solitamente nella pubblica amministrazione le idee su cosa debba essere un indicatore sono poche, confuse e gestite male. Calcolare un indicatore è sempre qualcosa molto meno banale di quello che sembra: generalmente abbiamo indicatori molto sofisticati e capirne il significato e le implicazioni è molto complicato.

3.2 Analisi multidimensionale

La teoria che c'è dietro il data warehouse è di natura geometrica: parliamo di analisi multidimensionale, pattern recognition (i.e., concetto di causalità e correlazione o effetto), Business Intelligence e machine learning.

Possiamo fare una proiezione per rappresentare un cubo su un piano, quindi avremo l'immagine 2D di un oggetto 3D. Uno spazio di dimensione n può contenere solo oggetti di dimensione $\leq n$. Il problema delle proiezioni è la perdita di informazioni, d'altra parte permette di continuare a rappresentare almeno una vista di un oggetto a n dimensioni su uno spazio a k dimensioni con $n > k$. Se non volessimo perdere informazioni, dovremmo rappresentare tutte le viste (cioè un numero minimo che ci consenta di avere tutte le informazioni): così facendo potremmo ricostruirlo esattamente.

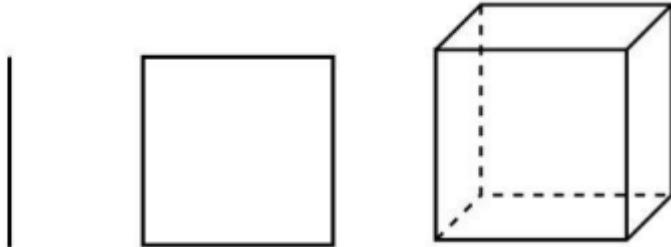


Figura 3.2: Varie dimensioni

Chiarito il concetto di proiezione, possiamo anche fare proiezioni di ordine superiori, ad esempio un ipercubo sul piano della lavagna. Per farlo cominciamo a disegnare il cubo di partenza e poi, sapendo dalla fisica che lo spazio tempo è in 4 dimensioni, lo replicheremo tante volte quante ne occorrono per rappresentare la sua traiettoria. Rappresento quindi il cubo e tutte le sue posizioni temporali al tempo t .

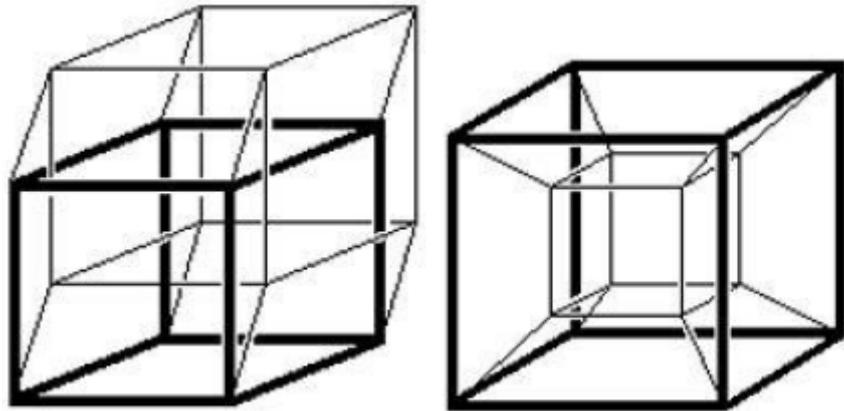


Figura 3.3: Ipercubi multidimensionali

Allo stesso modo possiamo disegnare un ipercubo disegnando delle diagonali che si allontanano dai vertici del cubo: consideriamo uno spigolo solido e tracciamone un altro che si allontana dal centro dell'oggetto: replicando questa operazione per tutti gli spigoli, otterremo un cubo che si gonfia. Bisogna immaginarlo come un oggetto solido a 4 dimensioni pensando a n repliche del cubo di dimensione $\{x, x + \delta, x + 2\delta\}$ e così via: lo spazio intermedio è pieno con una densità a 4 dimensioni, ci sono più copie del cubo contemporaneamente nello stesso posto.

A questo punto possiamo astrarre il concetto di rappresentazione di un punto geometrico e riferirlo non più allo spazio dei punti ma a quello dei dati. Ad esempio immaginiamo il solito database cliente-acquista-prodotto. Cosa significa dire cliente? Significa considerare uno spazio lineare di clienti dove i punti potenzialmente possono assumere qualsiasi valore possibile. Quindi immaginare uno spazio lineare in cui ordinare secondo un certo criterio i clienti significa rapportare un tipo di entità a uno spazio lineare su cui vale una relazione d'ordine definita tramite l'id. Ogni id identifica univocamente un cliente ed è un oggetto definito su uno spazio lineare dei numeri interi.



Figura 3.4: Cliente Acquista Prodotto

Cosa significa dire cliente-acquista-prodotto? Se abbiamo lo spazio dei clienti e lo spazio dei prodotti, la relazione è una coppia cliente-prodotto: avremo quindi un punto all'interno di uno spazio ad n dimensioni che rappresenta i fatti di interesse del database. Così abbiamo trasformato la teoria dei diagrammi entità-relazione in una teoria di geometria sugli spazi astratti dei dati. Questa è la teoria multidimensionale dei dati. È interessante perché riusciamo a trasformare operazioni di analisi sui dati in operazioni di analisi sulle figure. Il problema è che in un database dove ci sono 10 tipi di entità dobbiamo lavorare in uno spazio a 10 dimensioni e inoltre, se ci sono tante relazioni, non basterà un punto per rappresentare un fatto, ma molti tipi diversi di punti che legano queste dimensioni. Quando combiniamo i punti attraverso i tipi di relazione otteniamo delle n-ple che rappresentano i fatti di interesse all'interno dello spazio multidimensionale.

→ Fare una query su database significa estrarre un oggetto da uno spazio a n dimensioni.

Consideriamo ora la relazione prodotto-negozi-data (relazione ternaria di vendita) in uno spazio a n dimensioni. Questi oggetti non sono dei tipi di entità, per il momento possiamo chiamarli database. In tal modo abbiamo un database con tutti i prodotti, uno con tutti i negozi e uno dove ci sono tutte le date in cui è stata effettuata una vendita.

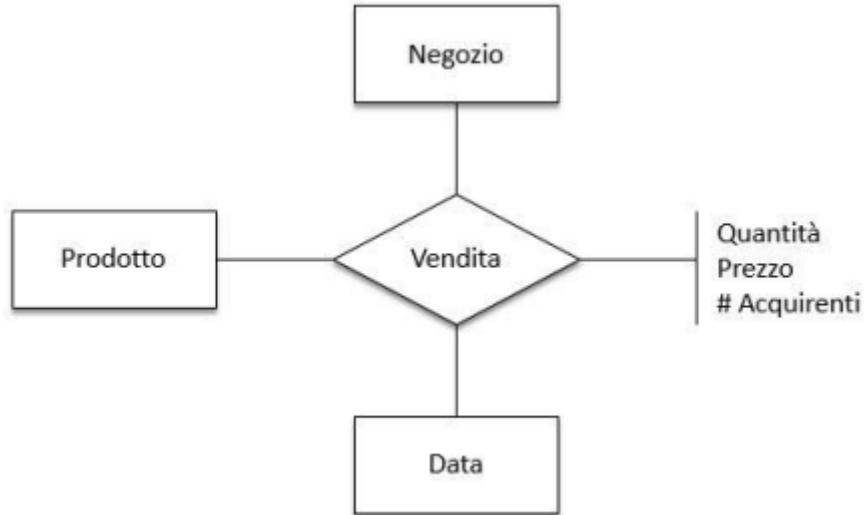


Figura 3.5: Vendita FACT

All'interno di questo spazio consideriamo un cubo, questo rappresenta una data di inizio e una di fine, un prodotto di inizio e uno di fine, poi un negozio di inizio e uno di fine. Fare le query sul database significa tagliare a fette e poi a cubetti (slice and dice) il cubo di origine, dove ogni cubetto rappresenta un fatto elementare o un'istanza di questa relazione. Per ogni fatto andiamo a rappresentare sul diagramma entità relazione la quantità, il prezzo, il numero degli acquirenti (dato che il database è analitico e dobbiamo fare l'operazione di aggregazione ci serve contare gli utenti per n transazioni andando a mettere dei valori unitari per ogni occorrenza).

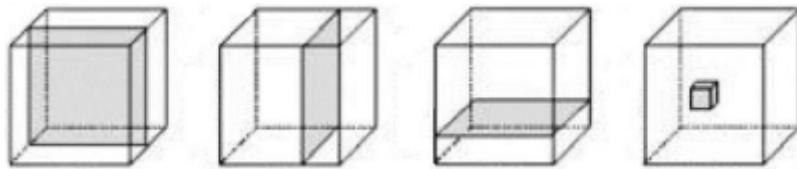


Figura 3.6: Operazioni sui cubi

Questi oggetti prendono il nome di misure all'interno dello spazio e possiamo utilizzarli per avere tanti tipi diversi di relazioni. La finalità è quella di fare operazioni sintetiche a livello alto della piramide di Anthony.

3.2.1 Operazioni concettuali

Consideriamo ora uno spazio a n dimensioni. Le operazioni concettuali che posso effettuare su questo spazio sono:

- La prima operazione che possiamo fare è quella di espandere lo spazio del punto prendendo una sfera o ridurre la stessa fino a farla diventare un punto. Se aggreghiamo tutti i fatti attorno ad una certa coordinata, sapremo, riferendoci all'esempio di prima, quanti pezzi sono stati venduti di tutti i prodotti, i pezzi usciti dalla cassa e il prezzo medio. Possiamo aggregare lungo una certa dimensione, non considerando il fatto elementare ma la somma, media o un qualunque operatore di natura statistica. A questo punto espandendo lungo una certa dimensione non consideriamo più il fatto elementare, ma il fatto aggregato. Questa operazione di espansione si chiama “drill down” (espressione che viene dal mondo del data mining: qui si immagina che il database sia una miniera mentre noi siamo i minatori che scopriamo diamanti corrispondenti alle informazioni). Nello specifico, abbiamo un prisma aggregato e lo spezzettiamo nei sotto cubi finché troviamo il fatto elementare.

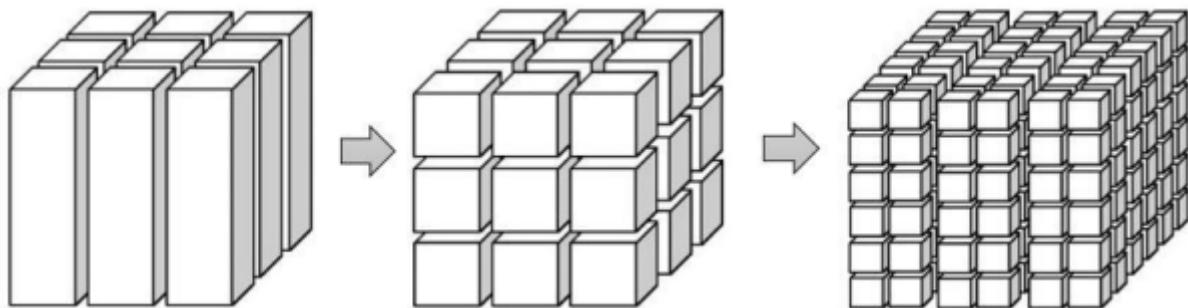


Figura 3.7: Operazione di DRILL-DOWN

Alternativamente abbiamo l'operazione di “roll up” o di aggregazione (invece di vedere tutti i valori analitici osserviamo il valore sintetico). Queste due sono le stesse operazioni percorse però nelle direzioni opposte.

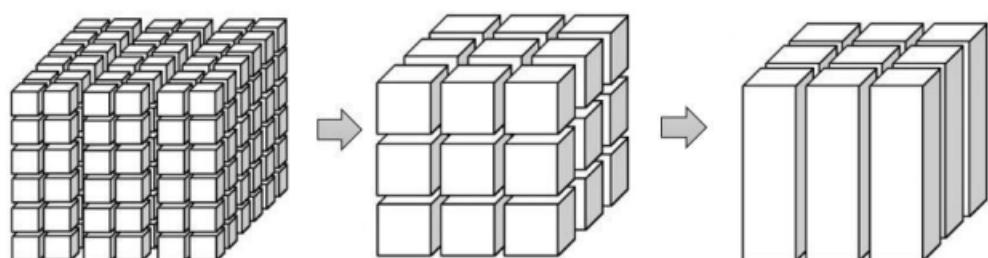


Figura 3.8: Operazione di ROLL-UP

- La seconda operazione è lo “Slice and dice”, ossia prendiamo un cubo, lo tagliamo a fette e successivamente a dadini (eventualmente quei dadini li dividiamo in sottocubi): un cubo aggregato è tutta l’informazione, invece i dadini sono paragonabili ad una query;
- La terza operazione è il “Pivot”, in cui invertiamo due assi cartesiani (scambiando la x con la y) oppure in uno spazio a n dimensioni ruotiamo il sottospazio. Di solito abbiamo $y = f(x)$, x è la variabile indipendente e y è dipendente. Quindi avremo $x = f^{-1}(y)$. Invertendo gli assi possiamo capire se rappresentare il fenomeno: in questo modo fornisce una relazione di causa effetto e potremmo ricondurci eventualmente ad un pattern ricorrente (lo strumento per effettuare pivot su Microsoft è Excel interfacciabile su Analisys Services, mentre su Pentaho è Mondrian).

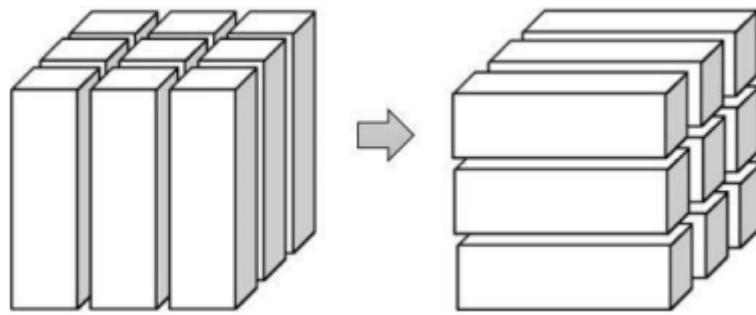


Figura 3.9: Operazione di PIVOTING

Giuseppe D’Amuri
Federico De Luca
17/11/2016

3.3 Analisi Multidimensionale e Data Warehousing

In realtà nel multidimensionale si utilizzano molti concetti dei normali DB. Ritroviamo la metodologia tipica di progettazione di un DB, che viene suddivisa in: progettazione a livello concettuale, quella a livello logico e quella a livello fisico. Nei DW si utilizza un’architettura a tre livelli (Three-tier).

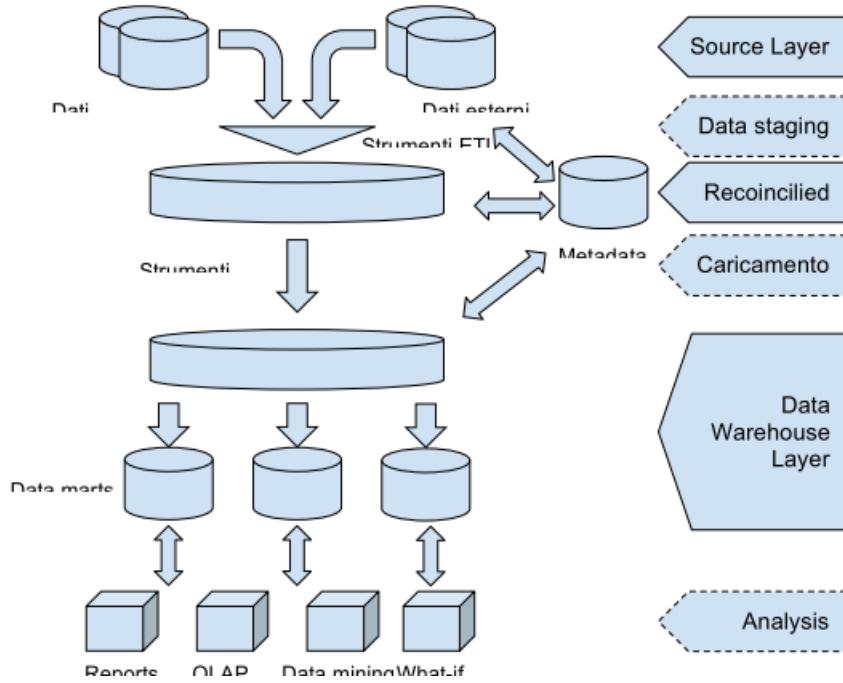


Figura 3.10: Architettura Three-Tier per i DW

In uno scenario classico le diverse organizzazioni o compartimenti dialogano effettivamente tra di loro, ma non hanno le stesse informazioni. Per questo nel mondo reale prima di tutto è necessario eseguire la cosiddetta operazione di Data Integration. Non vi è un mero problema di diversità fisica: il vero problema è che questi dati possono essere sia espressi che interpretati diversamente. Se combinati opportunamente però, questi dati alla fine possono risultare molto esplicativi ed utili, in grado di risolvere molti problemi di ambiguità o mancanza di informazione se visti isolatamente. All'interno di un DB isolato i dati sono tutti consistenti tra di loro. Noi invece abbiamo bisogno di effettuare una collezione di questi pezzi e renderli consistenti anche a valle della combinazione, risultando quindi alla fine sovrapponibili. Nel primo livello dell'architettura, denominato Source Layer abbiamo tutti i dati provenienti da differenti sorgenti, non necessariamente DB, in un unico DB che è chiamato Schema Riconciliato, facente parte del secondo strato dell'architettura denominato Reconciled Layer. I dati che a monte giungono qui provengono quindi da differenti ambienti, e vengono diversificati in Dati Operazionali e Dati Esterni. I primi provengono tipicamente da DB visti sino ad ora, e concretizzano le operazioni transazionali CRUD. I dati esterni invece possono provenire da varie fonti. Un esempio di informazione interessante che si potrebbe ricavare consultando i Dati Riconciliati è la flessione delle vendite, che oltre a considerare soltanto le vendite di veicoli, questi dati devono essere incrociati magari anche con l'aumento del prezzo del petrolio, oppure con i cambi delle varie valute etc.. Quindi non abbiamo bisogno soltanto di informazioni Operazionali, ma anche di quelle Esterne. Inoltre troviamo qui anche i Metadati che devono essere presi in considerazione; essi servono a contestualizzare i dati che provengono da un certo ambiente. Ad esempio, in concomitanza con i dati sulla vendita, potrebbe essere utile riportare la procedura di ordinazione utilizzata dagli utenti in una determinata vendita. Sostanzialmente sono delle informazioni ausiliarie, di contesto, che però devono essere messe assieme alle precedenti per ottenere un maggiore comprensione del fenomeno. Nel passaggio dallo strato dei Dati Riconciliati al Data Warehouse vero e proprio, che fa parte dell'ultimo strato della Three-tier per i DW,

avvengono le cosiddette operazioni ETL (Extraction, Transformation and Loading), ovvero di estrazione, di trasformazione e di caricamento. Nella descrizione di questa fase è bene tenere presente che a valle della procedura di Data Staging, ovvero di raccolta dei dati dalle varie fonti e nell'immissione nello strato dei Dati Riconciliati, i dati non sono puliti e non sono completi a volte. Tipicamente prima di essere immessi nel DW vero e proprio devono essere firmati da un responsabile e devono obbligatoriamente essere verificati. Una volta che i dati sono arrivati quindi, bisogna effettuare delle verifiche di consistenza e di validità/solidità, e solo dopo vengono immessi nel DW. Il DW può contenere tutta la storia della mia azienda. Mentre i Dati Riconciliati risiedono su un DB tipicamente relazionale, quelli nel DW sono immagazzinati in uno schema multidimensionale. Naturalmente la quantità dei dati presenti nel DW è molto più grande! In grandi aziende si scrivono GB di dati solo come LOG, è auspicabile aspettarsi quindi un paio di ordini di grandezza in più per i dati immagazzinati veri e propri. Il DW è quindi un oggetto che deve essere ottimizzato. Si tenga presente che nel multidimensionale non ci sono JOIN, ma sono molto prestanti le operazioni di aggregazione. Per raggiungere comunque delle performance ottimali, bisogna suddividere il DW, che di per sé è un vero e proprio ipercubo di dati, in sottocubi tematici denominati Data Mart. Ognuno rappresenta soltanto una parte dei dati racchiusi nel DW, diversificati per Area Tematica. In genere si progettano direttamente i Data Mart, anche se tecnicamente i dati alla fine devono comunque essere messi insieme. A questo punto abbiamo i quattro mondi più diffusi per effettuare Analisi Dati: abbiamo i Report che sono uno strumento cartaceo dettagliato di analisi. Un esempio potrebbe essere la Dichiarazione dei Redditi od il Bilancio Annuo che le aziende sono tenute a fare. Consistono sostanzialmente nella storicizzazione certificata del DB in un certo giorno. Esistono degli appositi framework, uno dei quali è BIRT (Business Intelligence Report Tool), che è open source e fornisce come output un documento in molti formati disponibili. Anche in Word esiste la possibilità di creare un report e collegarlo ad una o più tabelle del DB. Permette una formattazione più efficiente rispetto ad Access. Ma si tenga presente che questi documenti vanno alla fine stampati. Il mondo della reportistica è quindi un mondo su carta essenzialmente. Vi è una notevole differenza tra dati al computer e dati su carta: vi è tutto un discorso di responsabilità; Poi abbiamo gli strumenti OLAP (Online Analytics Processing), che permettono una navigazione sui vari cubi. Su di essi si può effettuare la ricerca di correlazioni e contemporaneamente navigarli alla stregua di normali pagine web. Sono sostanzialmente delle applicazioni navigazionali; Un altro strumento è il Data Mining, che si compone a sua volta di tecniche di Business Intelligence e Decisions Support Systems tipo il Pattern Recognition, Clustering, Expectation/Maximization (EM), etc. Grazie a questi strumenti oggi con i Big Data si va verso una prospettiva scientifica molto differente. Si possono acquisire i dati di tutto il mondo e capirne il funzionamento semplicemente attuando queste tecniche, ovviamente facendole eseguire da un calcolatore. Si citino anche come importanti tecniche il Pattern Matching ed il Machine Learning; poi abbiamo anche strumenti What-If, che permettono di estrarre un modello matematico rappresentativo del fenomeno, che può simulare ad esempio il funzionamento di un'azienda. È in realtà un mondo nato da Excel, tanto è vero che in Excel è possibile fare un'estrapolazione prendendo l'andamento in un intervallo proiettandolo in avanti. Si possono effettuare delle Analisi Variazionali di particolari fenomeni variandone i parametri. Oggi ci sono però degli scenari più complessi oltre al già citato cosiddetto Business Game. Ci possono essere delle reti di Business Game collegati alla previsione di andamenti di differenti aziende andando ad incrociare i vari andamenti, che magari si influenzano reciprocamente. Ma non si simula solo il business in realtà. Questi strumenti sono utilizzati anche nell'ambito dello studio dei Cambiamenti Climatici. Esistono delle tecniche apposite di proiezione di tanti scenari eterogenei con modelli differenti. Tutto questo mondo si appoggia sui DW.

Riguardo l'ETL, abbiamo che l'Estrazione può essere Statica, nella quale i dati vengono

accumulati una sola volta, oppure incrementale o dinamica, ove si può scegliere di inserire i dati per l'appunto incrementalmente, per esempio quelli riguardanti l'ultimo mese. Riguardo la PULITURA, una fase preliminare alla trasformazione vera e propria, ma che può essere inglobata in quest'ultima, essa si esplosa in eliminazione di dati duplicati: a volte possiamo avere degli interi dataset duplicati; risoluzione dell'inconsistenza dei dati o tra dati diversi. Possiamo andare incontro a situazioni di dati mancanti, che in questa fase devono essere opportunamente gestiti. Per ognuna di queste esistono delle tecniche apposite, tipo algoritmi Minimum Edit Distance o basati sulla distanza di Hamming tra stringhe, tecniche tutte inglobate nella Data Cleaning oppure Data Cleansing. La fase di Trasformazione vera e propria si compone invece delle fasi di Aggiunta Metadati e di Normalizzazione ove si indicizzano opportunamente i vari campi del dato, di Standardizzazione ove si portano i dati secondo convenzioni comuni e successivamente quella di Correzione, ove si correggono eventuali dati sbagliati. Esiste il problema del carico di lavoro: a rigore ogni qualvolta si alimenta il DW bisognerebbe ricalcolare gli indici. Per questo motivo un DW viene alimentato tipicamente ogni due/tre mesi. Si effettuano quindi dei report bimestrali/trimestrali.

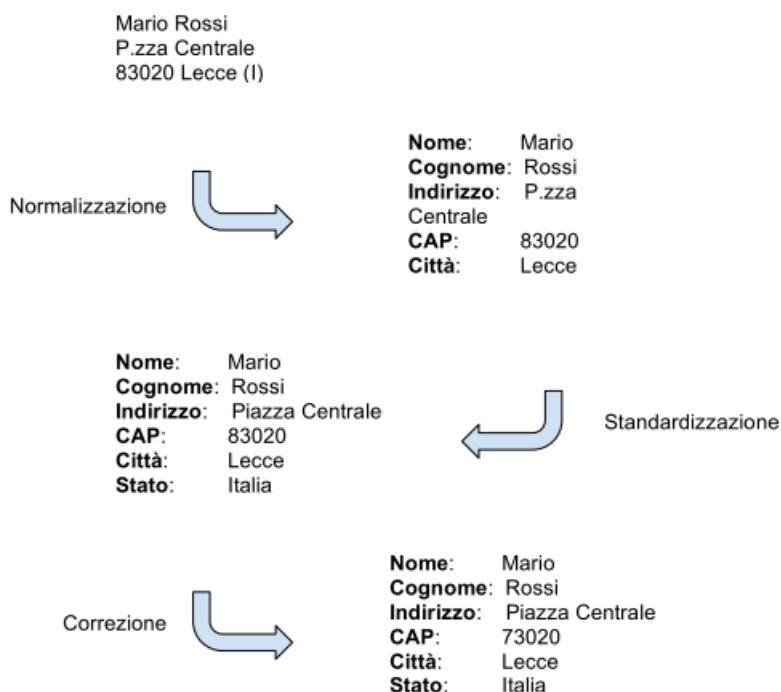


Figura 3.11: ETL: Normalizzazione, Standardizzazione e Correzione

Un DW è sostanzialmente un ipercubo di dati. Ognuno dei cubetti di cui si compone internamente il cubo rappresenta un Fatto Elementare (dato a granularità minima del DB multidimensionale). Tutti i DB multidimensionali presentano il problema della SPARSITÀ. Nel tipico scenario Cliente Acquista Prodotto, rappresentato opportunamente nel DW, un negozio potrebbe stare chiuso in una certa data oppure qualche prodotto/parte potrebbe non essere in vendita in un certo negozio. Bisogna gestirla opportunamente. I motori grafici utilizzano lo stesso approccio basato sulle matrici multidimensionali che si utilizzano nei DW. Una serie di fotogrammi non è nient'altro che la giustapposizione temporale di matrici 2D; ancora, la profondità di un oggetto a livello di singolo pixel è gestito dal cosiddetto Z buffer. La profondità

viene quindi associata ad un buffer multidimensionale. Un cubetto per noi è semplicemente un fatto che è accaduto o meno. Si utilizzano delle tecniche come il Pattern Matching/Recognition per trovare delle regolarità e classificare alcuni fatti. Si tratta semplicemente di individuare delle strutture geometriche che si ripetono. Tornando al problema della sparsità, se abbiamo tre dimensioni e 1000 elementi ad esempio, si arriva a dover gestire un oggetto dimensionalmente composto da 1000x1000x1000 elementi. Si arriva facilmente a numeri come 100^{25} quando si lavora ad esempio con 25 dimensioni da 100 elementi. Ma queste matrici non sono DENSE ma SPARSE! Se abbiamo elementi nulli (non popolati), non li memorizziamo! Vi è tutto un discorso sulla costruzione apposita degli indici per gestire queste situazioni. All'aggiunta di un nuovo fatto quindi, il processore dovrebbe ripercorrere tutto il DB. Si utilizzano quindi tecniche GPGPU per effettuare il parallelismo MIMD massivo ed utilizzare un numero elevato di processori in parallelo per processare questi dati.

Con le matrici multidimensionali le Query possibili sono essenzialmente di tre tipi: quelle di tipo SLICING, ottenute applicando la tecnica SLICE & DICE, ove genericamente si estraggono dei piani, dei cubi, delle colonne. Un'altra operazione molto importante è l'AGGREGAZIONE sulla base di Gerarchie Multidimensionali. Ogni tipo si differenzia in singolo prodotto. Vendita per CATEGORIA/TIPO/PRODOTTO. Tramite tecniche di ROLL-UP e DRILL-DOWN congiuntamente ad operazioni S&D si naviga attraverso le gerarchie multidimensionali. Un'aggregazione si compone quindi di ROLL-UP e DRILL-DOWN. Si effettuano quindi delle proiezioni, ove si proietta un qualsiasi ipercubo multidimensionale nello schermo 2D. Questa navigazione nella gerarchia multidimensionale è detta Navigazione OLAP. L'operazione di DRILL-DOWN mette in luce eventuali sparsità. Con l'operazione di PIVOTING effettuiamo ricerca di eventuali correlazioni, da non confondersi con le relazioni di causalità (causa-effetto) tra eventi. Una volta trovata un'eventuale correlazione non sappiamo chi è la causa e chi l'effetto!

Tecniche di Data Mining: sono delle tecniche che permettono di trovare delle regole associative. Ad esempio, l'implicazione vendita delle scarpe → vendita delle calze. Si introducono concetti come Supporto, Confidenza. Sulla base di tutte queste tecniche ed operazioni si effettua la cosiddetta MBA (Market Basket Analysis). Le tecniche di clustering sono molto importanti in questo. I dati vengono aggregati in certi gruppi, blocchi, bolle differenti, a seconda della terminologia in uso. Il Data Clustering viene facilitato dall'Analisi Multidimensionale. Il mondo OLTP è quello legato alle transazioni ed al CRUD, mentre il mondo OLAP è quello dei dati analitici, fattibile a diversi livelli. Quando si effettuano delle aggregazioni nel mondo OLAP si parla più propriamente di MOLAP (Multidimensional OLAP); ma anche in SQL abbiamo le operazioni di aggregazione. Quindi l'OLAP è possibile anche su DB operativi, ed in tal caso viene classificato come ROLAP (Relational OLAP). Anche a livello operativo è necessario ricordare la storia dei clienti. Si utilizzano quindi eventuali strumenti DSS di supporto alle vendite. Nella piramide DMO di Anthony troviamo non solo gli strumenti operativi, ma anche decisionali! Ma non solo a carico della direzione c'è bisogno di queste cose, ma anche in piccolo, nello strato transazionale/operativo quindi. In generale è bene sfruttare le interfacce per effettuare delle Query Proattive. La differenza di Google con Facebook è che su quest'ultimo le cose più importanti ce le dice già lui senza che interagiamo con esso! I compleanni degli amici ad esempio. Si effettuano le interrogazioni più importanti e le si propongono successivamente all'utente. Queste non sono altro che KPI, in piccolo, ed il loro ottenimento rappresenta un esempio di applicazione OLAP a livello di DB relazionale, operativo → ROLAP.

Ciclo di vita DW. Dietro la progettazione di un DW ci sono delle scelte strategiche importanti. Vi sono due approcci: Approccio Top-Down e Bottom-Up. Nel primo vi è un ordine di priorità. Sulla base di quello che il committente vuole, noi proponiamo e via via comuniciamo il costo, scendendo sempre più in dettaglio a partire dal problema. Questi approcci tipicamente sono congiunti ad una Ristrutturazione Aziendale che comporta la scelta di regole drastiche

e l'impiego di operazioni onerose. Queste tecniche forniscono i migliori risultati ma portano al dover scontrarsi con dei cambiamenti anche molto duri talvolta. Sono richieste quindi delle persone esperte: non è una questione di tecnologie. Non è solo il risultato a dover essere tecnicamente buono.

Poi abbiamo l'approccio Bottom-Up. Aproccio che parte da dei DB già fatti bene, dai quali possiamo estrarre degli indicatori (KPI) interessanti. Via via astraiamo dai dettagli di livello più basso per selezionare i vari elementi di cui si comporrà alla fine il DW.

DATA MART. Si parte da un primo Data Mart, ad esempio quello relativo al settore vendite, che tipicamente è il più prioritario. Dopo un certo tempo avremo il relativo Data Mart definitivo relativo alle vendite, ma nel frattempo possiamo lavorare in parallelo su Data Mart di altri settori.

Ricordiamo ora le varie fasi che compongono il ciclo a cascata della progettazione: Analisi e Conciliazione delle Sorgenti assieme all'Analisi dei Requisiti, Progettazione concettuale, Raffinamento del carico di lavoro, Progettazione logica, Progettazione dell'Alimentazione e progettazione fisica (DB). Si effettuano delle interviste al personale che dirige l'azienda, adottando un modello a Piramide o ad'Imbuto. L'obiettivo è comunque sempre lo stesso. Approccio a piramide (Bottom-Up) ed approccio ad imbuto (Top-Down). Così come nei DB utilizziamo dei Design Pattern, nel mondo DW abbiamo già delle soluzioni a problemi standard incontrati di frequente. Nei DW quindi abbiamo dei fatti significativi che esso deve contenere. Ma da quale DB vengono queste informazioni? Ad esempio potrebbe esserci un Data Mart apposito per il CRM (Customer Relationship Management) ove tipicamente si esercita la gestione dei reclami ad esempio. Bisogna scavare e trovare i settori dell'azienda all'interno dei quali queste informazioni sono rilevanti. È utile a tal proposito una tabella a singola entrata con due colonne: Data Mart, Fatti Rilevanti. In funzione del tipo dell'azienda si impone un certo tipo di progettazione.

Ora subentriamo in una specifica parte delle tecniche di progettazione. Nei DB ci sono gli ER. Qui abbiamo i DFM (Dimensional Fact Model), comunque collegati ai DB ma sono visti da un punto di vista differente. I DW si possono progettare a livello logico con gli schemi a stella. Modello Concettuale, Fisico e Logico sono sempre presenti. I Fact Model sono a livello concettuale. Essi servono per trasformare i fatti aziendali in tabelle multidimensionali. Ogni fatto ha un nome. In Cliente Acquista Prodotto abbiamo il fatto Vendita. La vendita come relazione è al centro. In una particolare configurazione DFM, la vendita potrebbe dimensionalmente essere associata ad una Data, ad un Prodotto e ad un Negozio. Questa potrebbe essere la reificazione o proprio la vera e propria relazione. In VENDITA ci mettiamo le misure. Il prezzo unitario è una caratteristica del prodotto, ma può essere riportato anche in VENDITA. Esiste anche un altro campo al quale in realtà non siamo abituati. Si chiama Numero Clienti. Valore unitario che si mette per agevolare le operazioni di conteggio/aggregazione. Molto spesso ci sono dei fatti che non hanno alcuna misura al loro interno. Ma il fatto che succeda e quando succeda è già significativo. Non possiamo fare una valutazione quantitativa! Un fatto esprime un'associazione molti-a-molti tra le varie dimensioni. Invece le dimensioni nel nostro caso sono data, prodotto, negozio. Immediatamente dopo aver definito il fatto, definiremo le Gerarchie Multidimensionali. Prodotto → Marca → Città. Questi successivi raggruppamenti definiscono delle gerarchie in una stessa dimensione. Difatti potremmo esser interessati alle vendite di un certo prodotto di una certa zona. Non mi basta sapere la marca, ma anche ad esempio se vi sono dei prodotti a Km/0. Si parla proprio di tendine nella navigazione di queste gerarchie. Navigandoci dentro per l'appunto, si compiono nient'altro che le già citate operazioni di Aggregazione che si compongono di ROLL-UP e DRILL-DOWN. Prodotto → Tipo → CATEGORIA. Il Negozio può ad esempio esser suddiviso per Responsabile delle Vendite. Le performance è perfettamente auspicabile che si suddividano in base al responsabile oppure al distretto delle

vendite. Ci possono essere delle aggregazioni non dimensionali bensì alternative. Un'altra aggregazione molto importante è quella sulle Date delle vendite. Le Date si compongono di vari campi elementari YYYY/MM/DD/H/M/S, i quali nel loro insieme formano la data. Potremmo avere dei raggruppamenti successivi per giorno o per settimane. Le settimane sono abbastanza importanti. In alcuni casi potrebbe essere molto significativo vedere gli eventi di interesse in base alla settimana. Abbiamo quindi Data → Settimana → Mese → Giorni di Vendita oppure Giorni Lavorativi. Queste ultime due sono delle Associazioni Ortogonalı tra di loro. Sono distinte, diverse. Una stessa data può essere di vacanza o lavorativa per differenti negozi. Ogni nodo prende il nome di Attributo Dimensionale. Le interrogazioni sono alla fine dei percorsi che tracciamo nel grafico dei fatti. Abbiamo una n-upla di attributi dimensionali. Queste sono le cosiddette Query MDX (Multidimensional Expressions). Non sono nient'altro che delle n-uple o tuple di attributi dimensionali. Ma come funziona l'RDB che sta dietro? Partendo da degli RDB si mettono insieme e si arriva a definire il cosiddetto Database Riconciliato. Qui tutte le gerarchie sono esplose in successive relazioni (1-n). Avremo ad esempio i tipi di entità DATA, che può essere di vacanza o lavorativa ed appartenere via via agli altri vari attributi dimensionali omogenei nella stessa entità. Nel database dovremmo inoltre tenere conto di dettagli relativi alla Località. Infatti una certa data può essere di vacanza per un certo stato ed essere invece lavorativa per un altro. Questi vengono detti attributi CROSS-DIMENSIONALI. Esattamente come ci si aspetta in un albero gerarchico, sono tutte relazioni (1-n). Le ridondanze non sono in questo caso molto preoccupanti. Tutti questi elementi sono candidati a divenire gerarchie multidimensionali. Qui noi vogliamo ottimizzare le query, le interrogazioni. Qualche ridondanza ci può ovviamente stare, e talvolta sarà proprio necessaria.

È importante comprendere la differenza tra DB operativi e quelli analitici. Alla fine le attività commerciali si strutturano sempre in vendite di prodotti e di servizi, ad esempio. Tipicamente esistono delle gerarchie standard abbastanza ricorrenti: le Gerarchie Spaziali, le Gerarchie Temporali. Inoltre avremo poi le Gerarchie Specifiche del problema. Sono alla fine dei concetti abbastanza logici ed intuitivi.

Come si fa invece a capire quali sono i fatti del mio RDB? Possiamo utilizzare un approccio Top-Down o Bottom-Up. Tipicamente i Fatti Significativi sono quelli che accadono più frequentemente. Il fatto cruciale sarà quello più frequente ed importante in assoluto: la VENDITA nel nostro caso. Non sarà DATA ad esempio, in quanto il calendario aziendale è già stabilito ad inizio anno. Non sarà PRODOTTO, in quanto il catalogo dei prodotti è già stabilito dall'azienda all'inizio dell'anno. Ci possono essere delle variazioni, ma non cambia nulla. La VENDITA è il fatto significativo, tra l'altro anche perché si aggiungono sempre, in base al principio di NO-DELETE. In tutto quindi quanti fatti ci sono? Due o tre sul cliente, in quanto le variazioni sul cliente sono locali e registrate nella relativa entità stessa. Un paio di fatti sul PRODOTTO. Tutti gli altri eventi sono a bassissime frequenze rispetto alle vendite. Distinguiamo quindi gli Eventi Primari da quelli Secondari. Esistono anche gli Attributi Opzionali, presenti solo nel qual caso ad esempio un Prodotto appartenga ad una determinata categoria (es. attributo dieta solo su prodotti di categoria alimentare). Possono anche esserci degli Attributi Descrittivi, i quali non servono propriamente a fare le operazioni di Aggregazione, quanto invece per avere informazioni più dettagliate su una certa dimensione di un fatto all'interno della sua relativa gerarchia dimensionale.

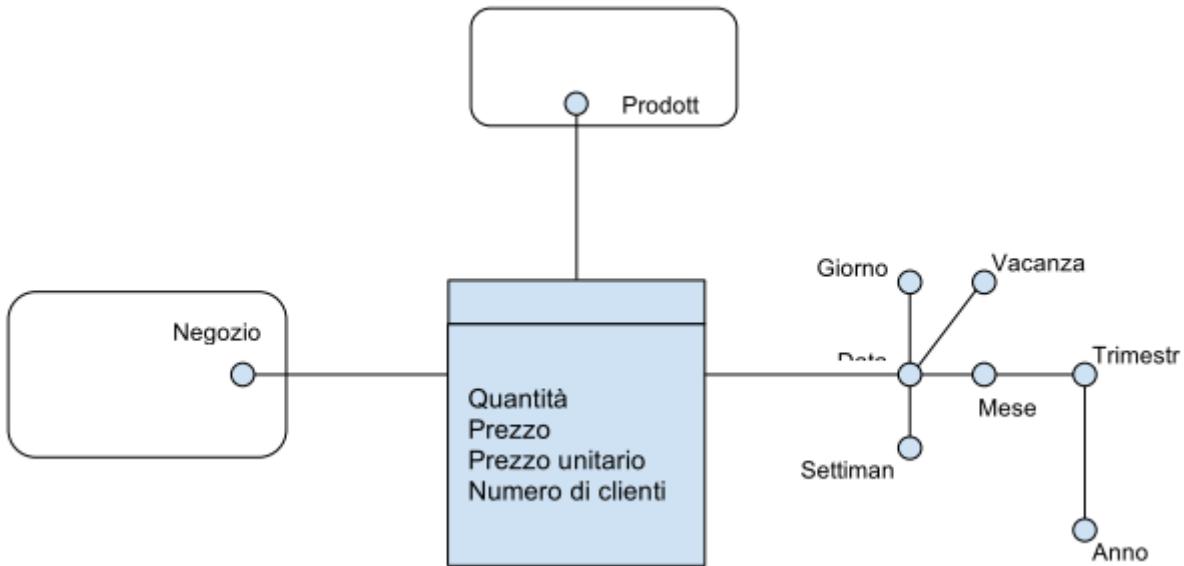


Figura 3.12: Dimensional Fact Model: DFM

Marco Chiarelli
Paolo Panarese
23/11/2016

3.4 COSTRUTTI AVANZATI DEL DFM

- **ATTRIBUTI CROSS-DIMENSIONALI:**

Sono attributi che dipendono da più dimensioni (appartenenti a diverse gerarchie). Un classico esempio è l'IVA: essa varia da stato a stato, ma anche in base alla categoria dei prodotti (i beni di prima necessità presentano un'aliquota IVA inferiore a quella dei beni di lusso).



Figura 3.13: Attributi Cross-Dimensionali

- **ARCO MULTIPLO:**

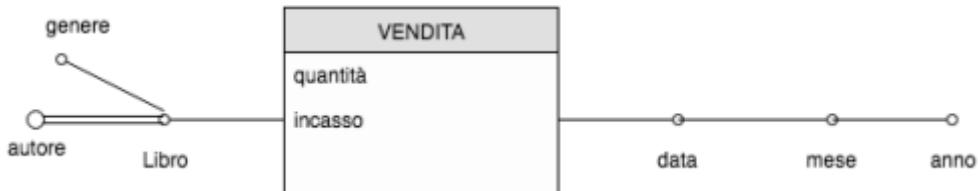


Figura 3.14: Arco Multiplo

Generalmente le associazioni fra attributi dimensionali sono 1:N. L'arco multiplo consente di rappresentare, invece, un'associazione N:M. In questo esempio si aggregano le vendite di libri in base agli autori. Un autore scrive più libri, ma un libro può anche essere scritto da più autori. Per questo motivo è necessario utilizzare un arco multiplo. Per dare maggiore consistenza alle aggregazioni tramite archi multipli, si devono attribuire dei **"pesi"** (ad esempio, un autore scrive un libro per il 90% e l'altro per il 10%; i pesi consentiranno di gestire le diverse percentuali degli incassi da attribuire ai due autori).

- **ADDITIVITÀ:**

Al fine di aggregare i fatti, è necessario definire gli operatori che lavoreranno sui valori delle misure. Queste ultime possono essere di tre tipi:

- MISURE DI FLUSSO: riferite ad un periodo di tempo, ad es. flussi di merce, di denaro, di cassa, il numero di prodotti venduti in un giorno, l'incasso mensile, il numero di nati in un anno;
- MISURE DI LIVELLO: valutate in un dato istante di tempo (numero di pezzi in magazzino);
- MISURE UNITARIE: valutate in istanti di tempo, ma in termini relativi (prezzo unitario di un prodotto, percentuale di sconto).

Esempio

Consideriamo una misura di livello: supponiamo che la mia azienda abbia in cassa 1000 euro nella sede di Lecce e 1000 euro in quella di Brindisi. La cassa, in totale, ha 2000 euro (posso sommare su gerarchie spaziali). Supponiamo, ora, che nella cassa di Lecce, alla fine della giornata di ieri, ci fossero 1000 euro, e che oggi ce ne siano 1000. In questo caso non posso sommare e affermare di avere in cassa 2000 euro! (la somma su gerarchie temporali non è consentita).

La tabella seguente indica quali operazioni sono consentite relativamente ai tipi di gerarchie:

	Temporali	Non Temporali
Misure di Flusso	SUM AVG MIN MAX	SUM AVG MIN MAX
Misure di Livello	Avg MIN MAX	SUM AVG MIN MAX
Misure Unitarie	Avg MIN MAX	Avg MIN MAX

Figura 3.15: Tabella Tipi di Misure

Una misura, dunque, si dice **ADDITIVA** su una dimensione se i suoi valori si possono aggregare lungo la corrispondente gerarchia con l'operatore di somma, altrimenti è detta **NON ADDITIVA**.



Figura 3.16: Inventario FACT

L'arco tratteggiato indica la “non additività”, e AVG e MIN indicano che l'aggregazione è consentita soltanto tramite gli operatori di media e minimo.

- **SCHEMI DI FATTO VUOTI:**

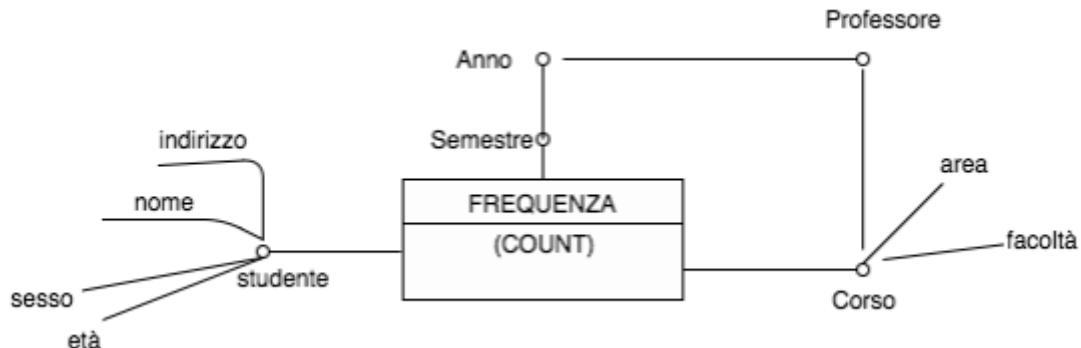


Figura 3.17: Empty FACT Schema

Sono fatti che non hanno misure. Servono principalmente a registrare il verificarsi di un evento, ad esempio la frequenza con cui uno studente segue un corso. Nel fatto Frequenza non troviamo misure, ma solo un contatore.

3.4.1 ESEMPIO DELLE VENDITE (DA E/R)

Al livello dei dati riconciliati, nel modello ER si devono esplicitare le gerarchie dimensionali e i fatti di business.

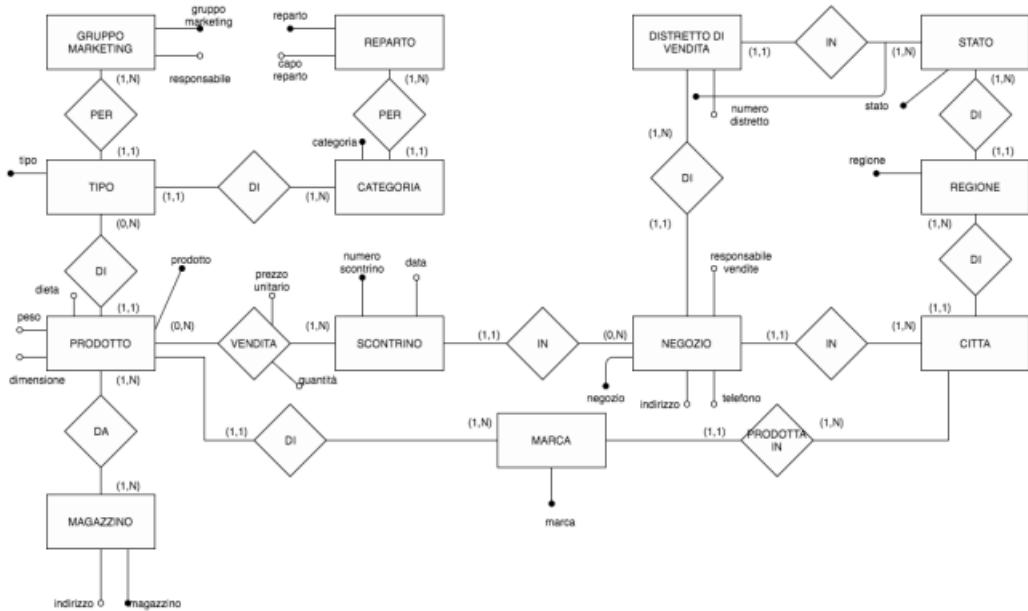


Figura 3.18: DB Riconciliato: VENDITE

I pallini scuri indicano che un particolare attributo è chiave primaria.

3.4.2 Passaggio dall'ER riconciliato al DFM: l'ALBERO DEGLI ATTRIBUTI

Il passaggio dal modello ER dei dati riconciliati al DFM avviene tramite un algoritmo di mapping, che mira a costruire il cosiddetto “**albero degli attributi**”, in cui la radice è il fatto, mentre ogni entità connessa al fatto è un nodo dell’albero.

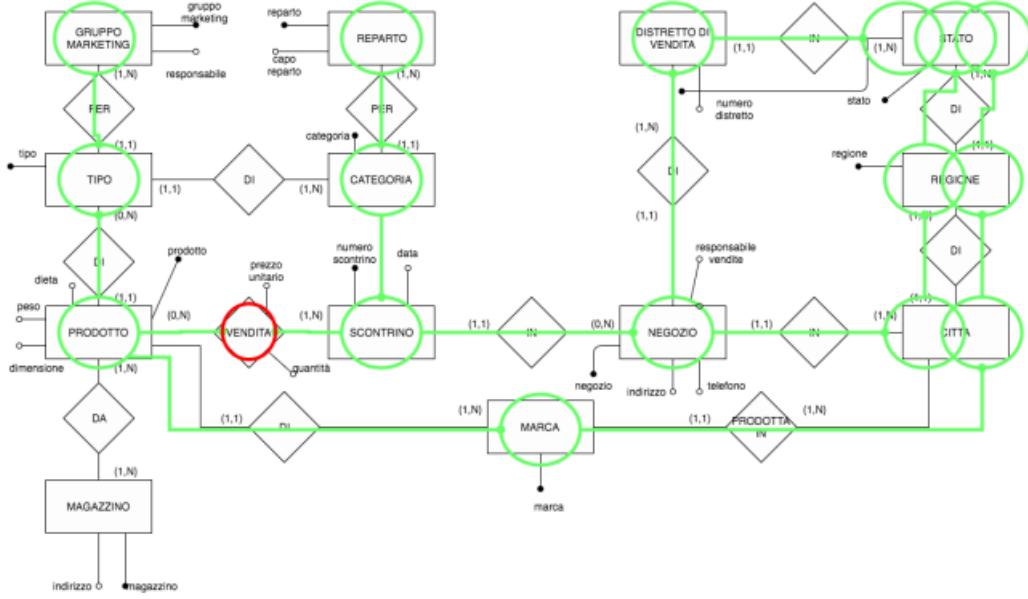


Figura 3.19: Individuazione Radice e Nodi su un DB Riconciliato

Individuati (figura sopra) la radice e gli altri nodi, si costruisce l'albero degli attributi:

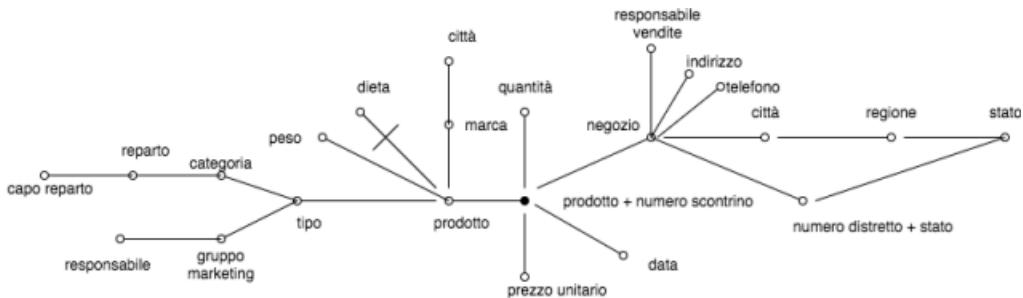


Figura 3.20: Albero degli Attributi

Sull'albero degli attributi si possono effettuare due operazioni:

- POTATURA: eliminazione di un nodo;
- INNESTO: elimino un nodo e collego i suoi figli direttamente al padre del nodo potato.

3.4.3 CARICO DI LAVORO

È un elemento importante da tenere in conto nella progettazione fisica del DW. È necessario stimare quanto incide l'inserimento dei nuovi dati e il ricalcolo degli indici sulle prestazioni del sistema.

3.4.4 PROGETTAZIONE LOGICA

SCHEMA A STELLA:

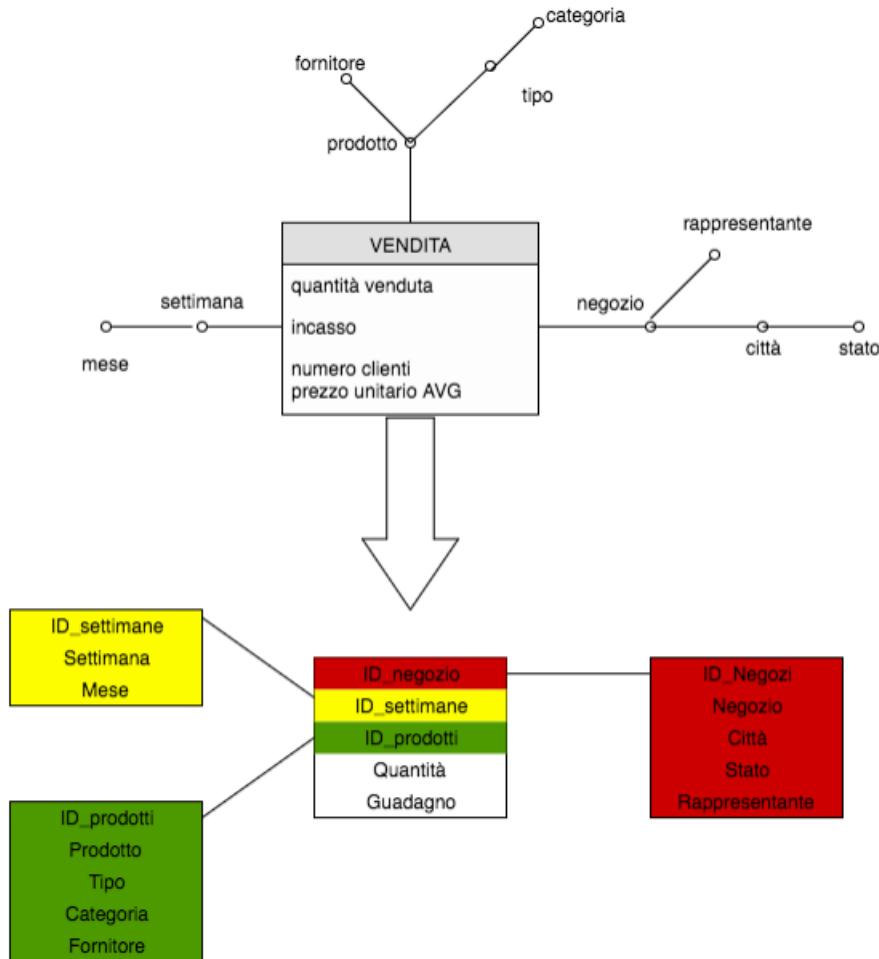


Figura 3.21: Passaggio da DFM a Star Schema

Lo schema a stella è composto da:

- **TABELLE DELLE DIMENSIONI:** una gerarchia dimensionale viene “compressa” in un’unica tabella. La chiave primaria della tabella è data dalla dimensione, mentre gli altri attributi sono costituiti dagli attributi dimensionali;
- **TABELLE DEI FATTI:** una tabella relativa a un fatto importa (come chiavi esterne) le chiavi di tutte le tabelle delle dimensioni, che insieme diventano chiave primaria della tabella del fatto.

Marco Mameli
Marco D’Amato
24/11/2016

3.5 LE TABELLE PIVOT DI EXCEL PER IL DATA WAREHOUSE

Consideriamo la seguente relazione quaternaria:

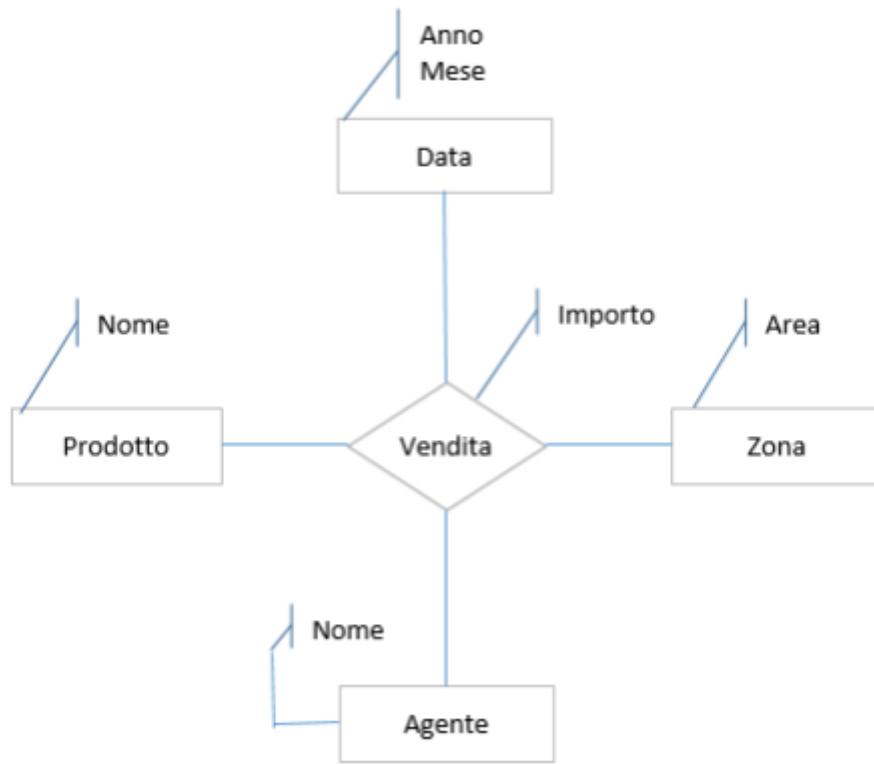


Figura 3.22: Relazione quaternaria VENDITA

Da tale diagramma concettuale il fatto più intuitivo e più frequente che si può estrarre è la VENDITA, che in realtà non è altro che la relazione tra le quattro entità. Quindi analizziamo le vendite in base al PRODOTTO, DATA, AGENTE e ZONA:



Figura 3.23: Vendita FACT

La gerarchia temporale completa è:

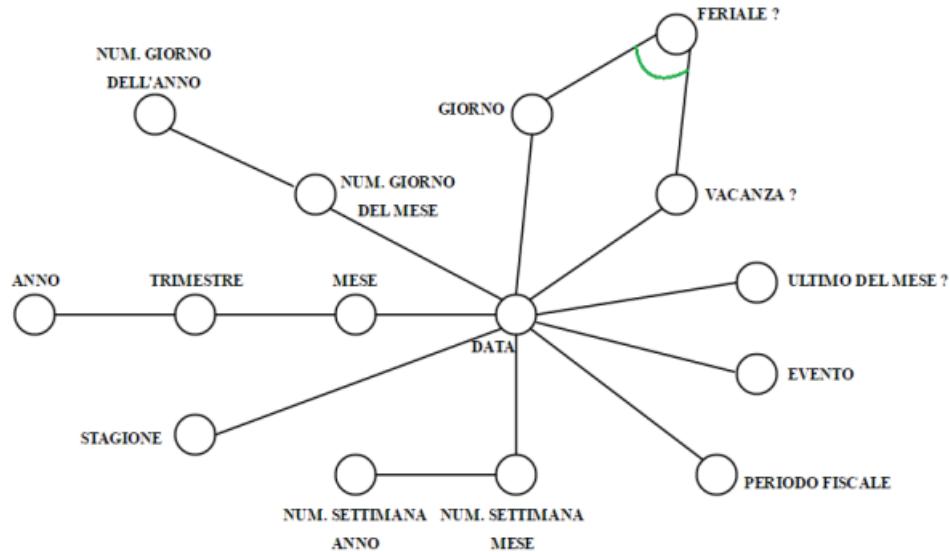


Figura 3.24: Gerarchia Temporale completa

In questo schema vengono indicate con “?” le variabili di tipo booleano. Dato il diagramma ER di partenza e supponendo che tutte le entità abbiano una chiave primaria, immaginiamo di voler fare una query che restituisca una tabella fatta in questo modo:

NOME PROD	ANNO	MESE	AREA	AGENTE	IMPORTO
...

Figura 3.25: Risultato tabellare di una Query

Se si volessero conoscere i prodotti inseriti, in Excel si può selezionare la colonna prodotti e una volta ricopierta, tramite la funzione “rimuovi duplicati” in “dati” che si trova sul pannello, si può vedere quali sono i vari prodotti distinti. La stessa cosa si può fare con le altre misure. Si osserva in questo caso che si hanno tre tipi di prodotto, due anni, diversi mesi, due agenti e due aree.

Andiamo a creare ora la tabella PIVOT, uno strumento analitico di reporting necessario alla creazione di tabelle riassuntive. Uno dei fini principali di queste tabelle è l’organizzazione di dati complessi tramite una scelta opportuna dei campi e degli elementi che devono comporla.

In Excel si seleziona l’intera tabella dei dati di partenza e si inserisce la tabella pivot andando in “inserisci” → “tabella pivot”. Viene creato un nuovo foglio di lavoro in cui è possibile creare tali tabelle riassuntive andando ad agire sui campi della tabella pivot alla destra del foglio.

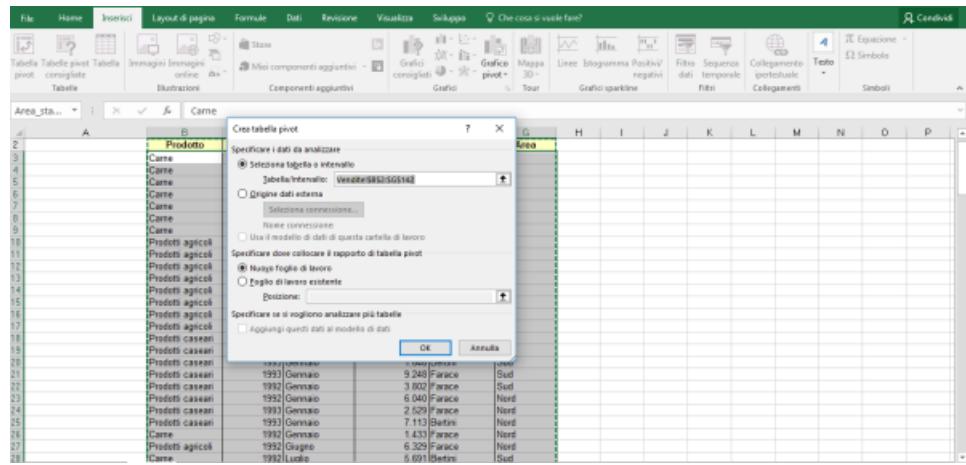


Figura 3.26: Microsoft Excel - Funzione tabella Pivot

Ad esempio, se si vogliono analizzare le vendite, si trascina il campo “vendite” nell’area “VALORI”, si può anche decidere il tipo di calcolo da utilizzare, come la somma, il conteggio, la media ecc. cliccando su “impostazioni campo valore”. Se si vuole la vendita dei prodotti distinta per anno, il campo “anno” lo si inserisce nell’area “COLONNE”. Ancora, continuando, si può aggiungere il campo “agente” in “COLONNE” in modo da avere una distinzione per agente in ogni anno.

Si può contemporaneamente calcolare la somma, il conteggio delle vendite ecc. semplicemente andando a inserire più volte il campo “vendite” in “VALORI” e selezionando le “impostazioni campo valore” opportune. Oppure, come si procede solitamente, si raggruppano più campi sulle “RIGHE”, in questo caso si può raggruppare “anno” e “mese”:

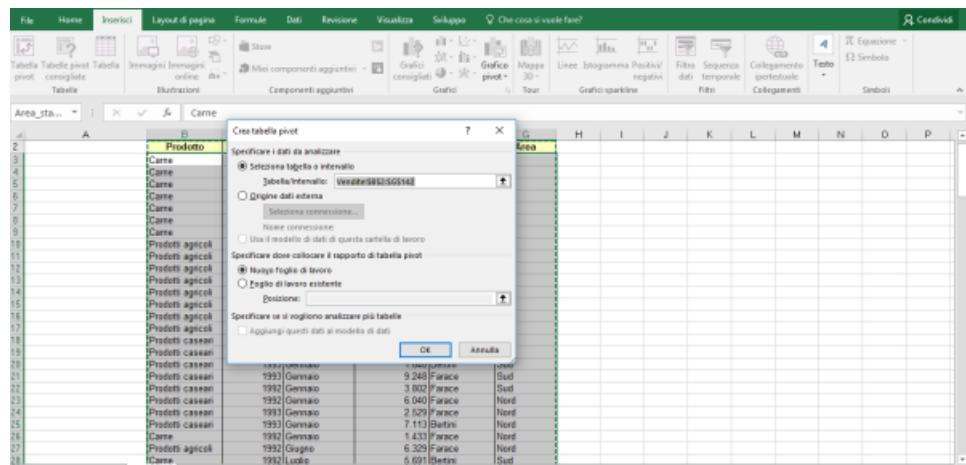


Figura 3.27: Microsoft Excel - Funzioni di Conteggio Somma - Tabella Pivot

OSS: In questo caso si è riscontrato un problema di data cleaning, nella tabella pivot compare due volte il mese di gennaio perché è stato riportato in maniera errata all’interno del sistema. Bisogna quindi effettuare tutte le operazioni di pulizia opportune durante la fase di ETL (Extract, Transform, Load). Abbiamo corretto il valore errato della tabella di partenza (cella D26) e aggiornato la tabella pivot: “analizza” → “aggiorna”.

3.5.1 QUERY

- Incremento percentuale delle vendite (importo) dal 1992 al 1993. COLONNE: Prodotto
 - RIGHE: Anno;
 - VALORI: Vendite (somma).

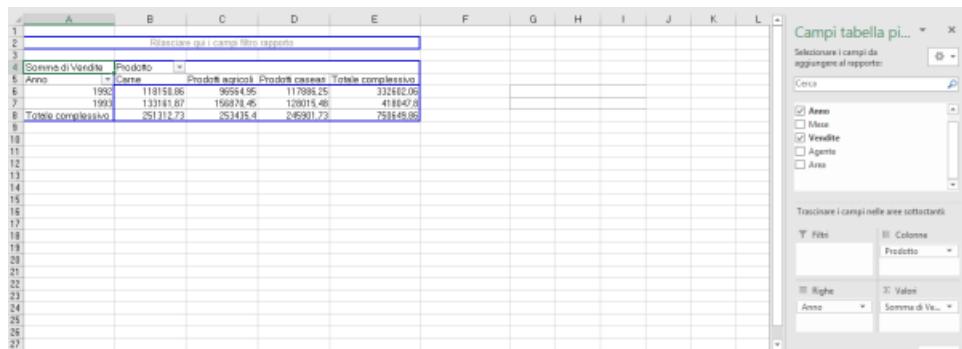


Figura 3.28: Microsoft Excel - QUERY Incremento percentuale Vendite

Per ottenere il valore in percentuale delle vendite in quel periodo si effettua prima la differenza tra il totale complessivo del '93 e del '92, il risultato lo si divide per il totale complessivo del '92 e lo si moltiplica per 100.

Per migliorare la precisione del risultato, nella sezione “numeri” in “home” si può andare a incrementare o decrementare le cifre decimali.

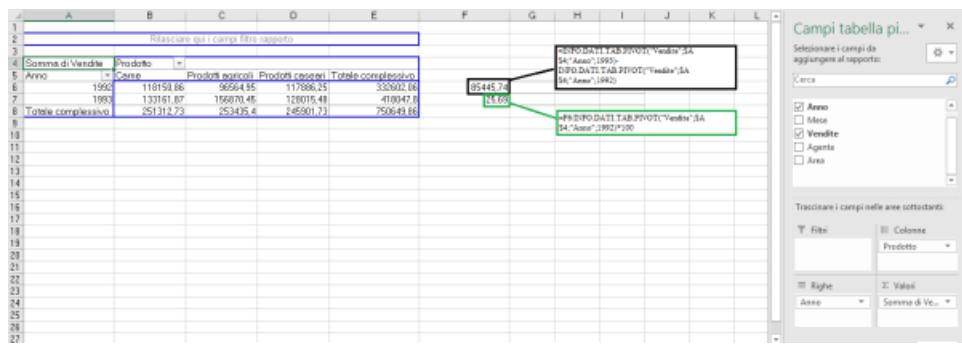


Figura 3.29: Microsoft Excel - Miglioramento precisione QUERY Incremento percentuale Vendite

- I due mesi in cui si è venduto maggiormente nel '92;
- I due mesi in cui si è venduto maggiormente nel '93.

RIGHE: Anno, Mese VALORI: Vendite (somma) Per verificare i 2 mesi in cui si è venduto di più si selezionano le colonne mese e totale, le si ricopiano e tramite la funzione

“ordinamento personalizzato” in “ordina e filtra” si andrà ad ordinare in modo crescente o decrescente (in questo caso crescente) in base ai totali. La stessa cosa la si fa con l’anno ’93. Si nota che nel ’92 i mesi in cui si è venduto di più sono luglio e maggio, nel ’93 gennaio e maggio.

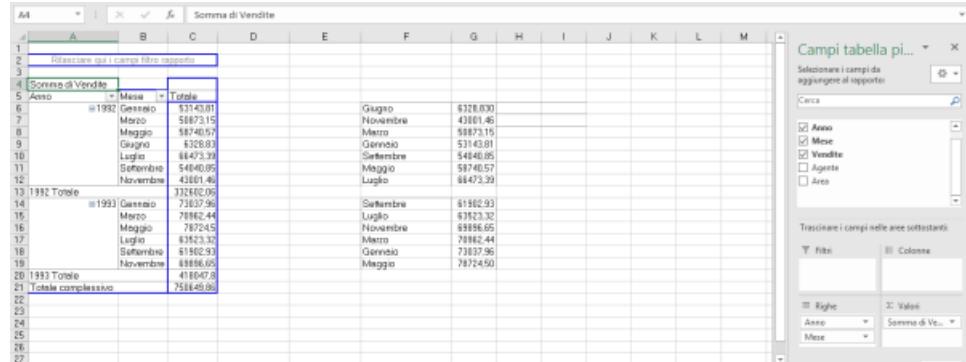


Figura 3.30: Microsoft Excel - QUERY Migliori mesi di Vendita nel ’92-’93

- L’area in cui si è venduto maggiormente nel ’92;
- L’area in cui si è venduto maggiormente nel ’93.
 - RIGHE: Anno, Area;
 - VALORI: Vendite (somma).

In entrambi gli anni l’area in cui si è venduto di più è il NORD;

- L’agente che ha venduto di più nell’area SUD nel ’93.
 - FILTRI: Area, Anno;
 - COLONNE: Agente;
 - VALORI: Vendite (somma).

In questo modo si può filtrare l’area e l’anno selezionando quelli richiesti. Per essere sicuri di quale sia il maggiore del valore totale si può utilizzare la funzione MAX tra i valori ottenuti, o andare ad ordinarli come visto in precedenza, in quanto i valori che si ottengono possono essere numerosi (e non 2 come in questo caso).

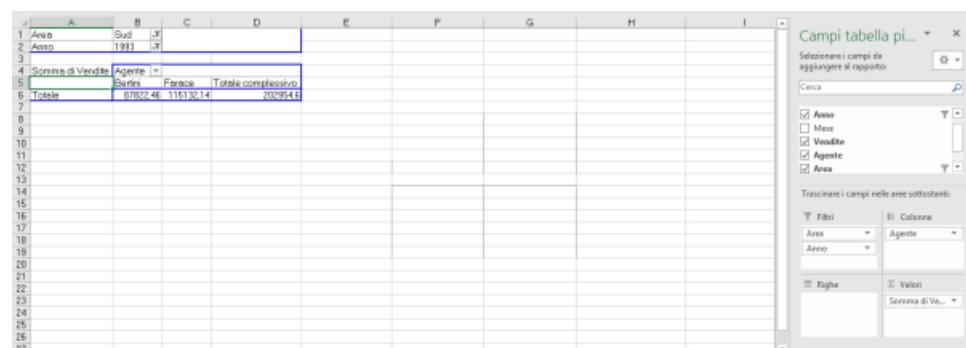


Figura 3.31: Microsoft Excel - Filtri PIVOT

L'agente che ha venduto di più è Farace;

- Il prodotto venduto maggiormente nel '92:

- FILTRI: Anno;
- COLONNE: Prodotto;
- VALORI: Vendite (conteggio).

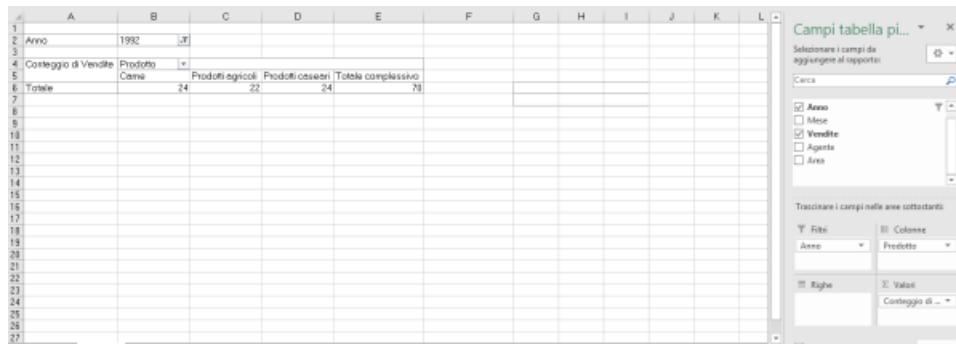


Figura 3.32: Microsoft Excel - QUERY Prodotto venduto maggiormente del '92

Selezionato l'anno '92, i prodotti che risultano più venduti sono la carne e prodotti caseari;

- Il prodotto che ha fatto incassare di più nel '92:

- FILTRI: Anno;
- COLONNE: Prodotto;
- VALORI: Vendite (somma).

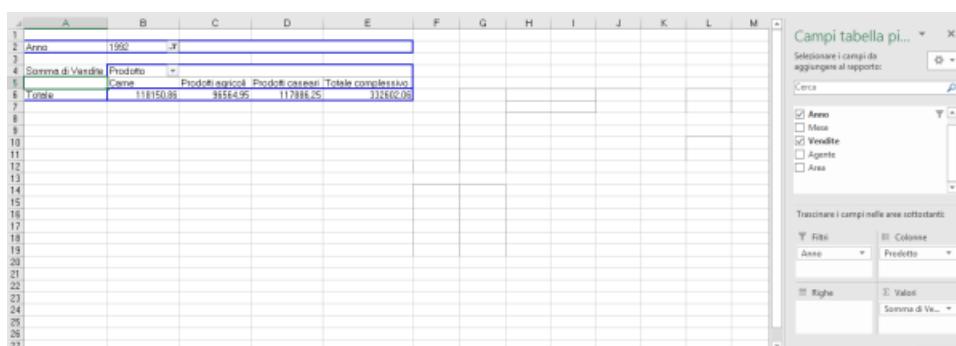


Figura 3.33: Microsoft Excel - QUERY Prodotto migliore dal punto di vista degli incassi del '92

Selezionato l'anno '92, il prodotto che ha fatto incassare di più è la carne. Per essere sicuri di quale sia valore maggiore del totale si procede come nella query 6;

- Prodotto venduto meno nel '92 nell'area NORD:

- FILTRI: Anno, Area;
 - COLONNE: Prodotto;
 - VALORI: Vendite (conteggio).

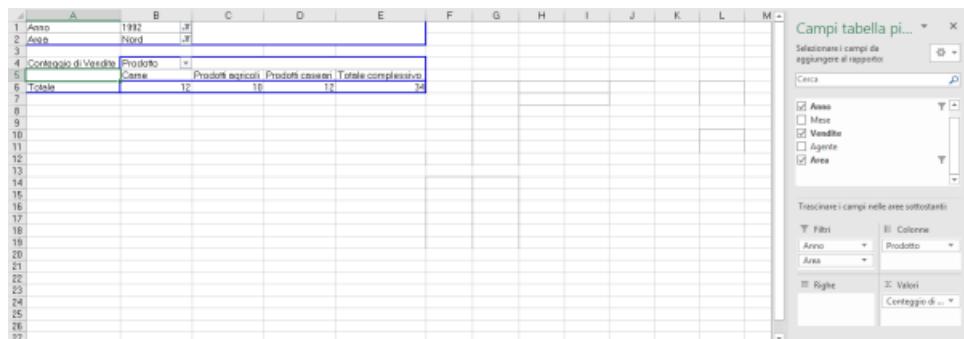


Figura 3.34: Microsoft Excel - QUERY Prodotto meno venduto nel '92 nell'area NORD

Selezionato l'anno e l'area richiesti, il prodotto che ha fatto incassare meno è “prodotti agricoli”. Anche in questo caso per essere sicuri di quale sia il minimo si può andare ad utilizzare le funzioni messe a disposizione da Excel (come la fx MIN o andando ad ordinare ecc.);

Immaginiamo ora di avere il seguente scenario, relativo ad un call center:

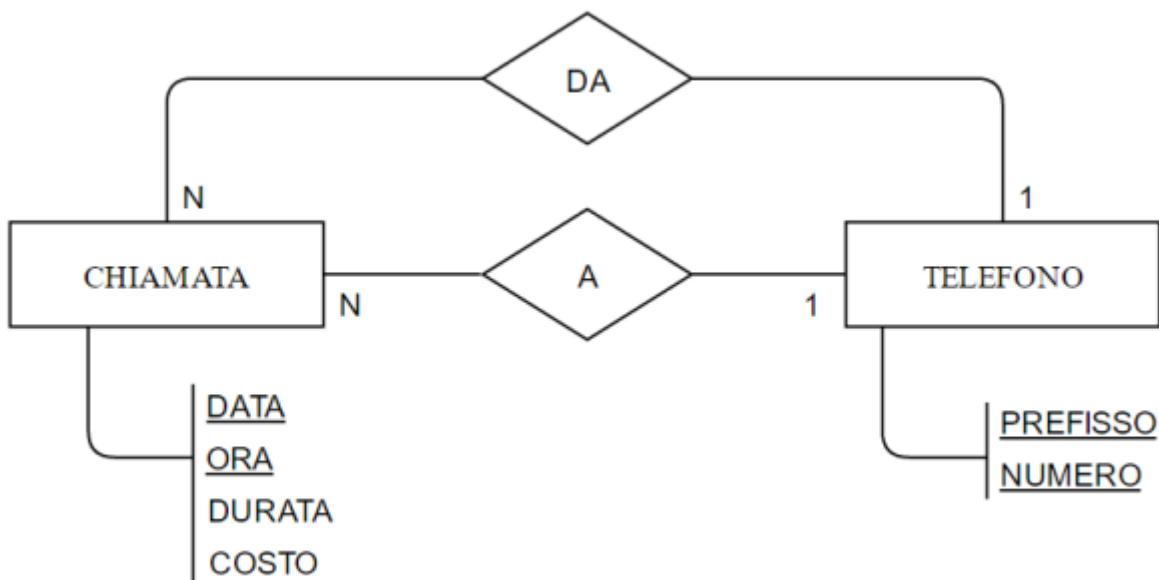


Figura 3.35: Call Center - ER Scenario

		CHIAMANTE		CHIAMATO			
ORA	DATA	PREFISSO	NUMERO	PREFISSO	NUMERO	DURATA	COSTO
...

Figura 3.36: Tabella per le query del precedente scenario

Il DFM associato è:

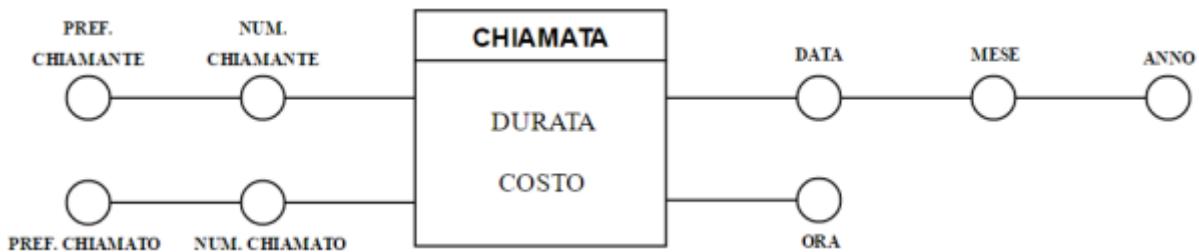


Figura 3.37: Chiamata FACT

Tale modello però risulta ridondante, cioè porzioni intere di gerarchie risultano duplicate come in questo caso. Utilizzando le gerarchie condivise si ottiene uno schema del genere:

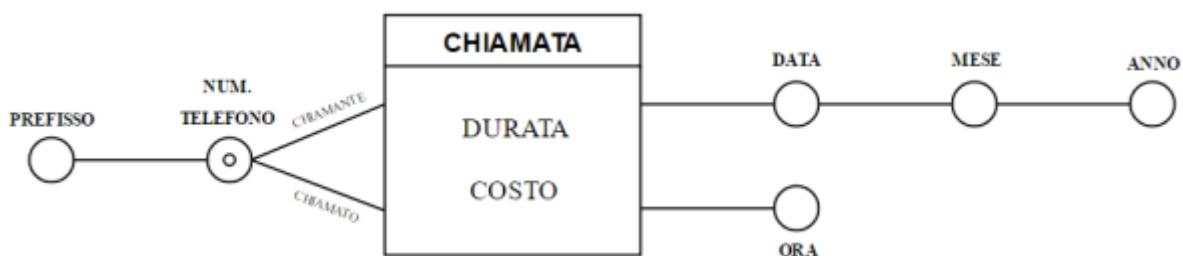


Figura 3.38: Chiamata FACT - DFM migliorato

QUERY:

- Numero di chiamate senza risposta nel mese di maggio 2016;
- Il prefisso con le telefonate più lunghe;
- Il mese del 2016 in cui sono state effettuate più chiamate.

3.6 PROGETTAZIONE LOGICA

3.6.1 Indicatori

Come abbiamo visto nelle lezioni precedenti, un passaggio importante per la creazione di un Data Warehouse è la revisione dei database con la creazione di date e gerarchie di aggregazione; le gerarchie di aggregazione possono essere di tipo temporale, spaziale oppure relative ad un dominio specifico. I Data Warehouse sono aggregazioni di dati multidimensionali in cui, a differenza dei normali database CRUD, sono presenti le operazioni di creazione e modifica, ma non di cancellazione. Perseguono quindi obiettivi diversi. Il modello concettuale di un Data Warehouse è il fact model: un fatto è qualcosa che avviene nel database o nell'azienda; nel primo caso si esegue un approccio di tipo bottom-up, nel secondo di tipo top-down. Si noti come nell'aggregazione dimensionale sia necessario tenere a mente la presenza di un lead time, ovvero il tempo di attraversamento intermedio tra l'attivazione di un processo causato da un evento e la sua conclusione, ad opera di un altro evento. Uno degli obiettivi di chi crea i database è l'estrazione di indicatori relativi ai tre tipi di risorse (Umane, Materiali, Immateriali). Ad esempio, nel caso di un'università, tali indicatori possono essere:

- Risorse Umane: livelli di cassa, flusso in uscita (costo risorse umane), numero medio di esami al mese per professore;
- Risorse Materiali: valore fabbricati, costi studenti, valore strumentazione;
- Risorse Immateriali: brevetti, società affiliate.

Schema a stella: È qui riportato lo schema a stella del fatto Acquisto per una catena di supermercati:

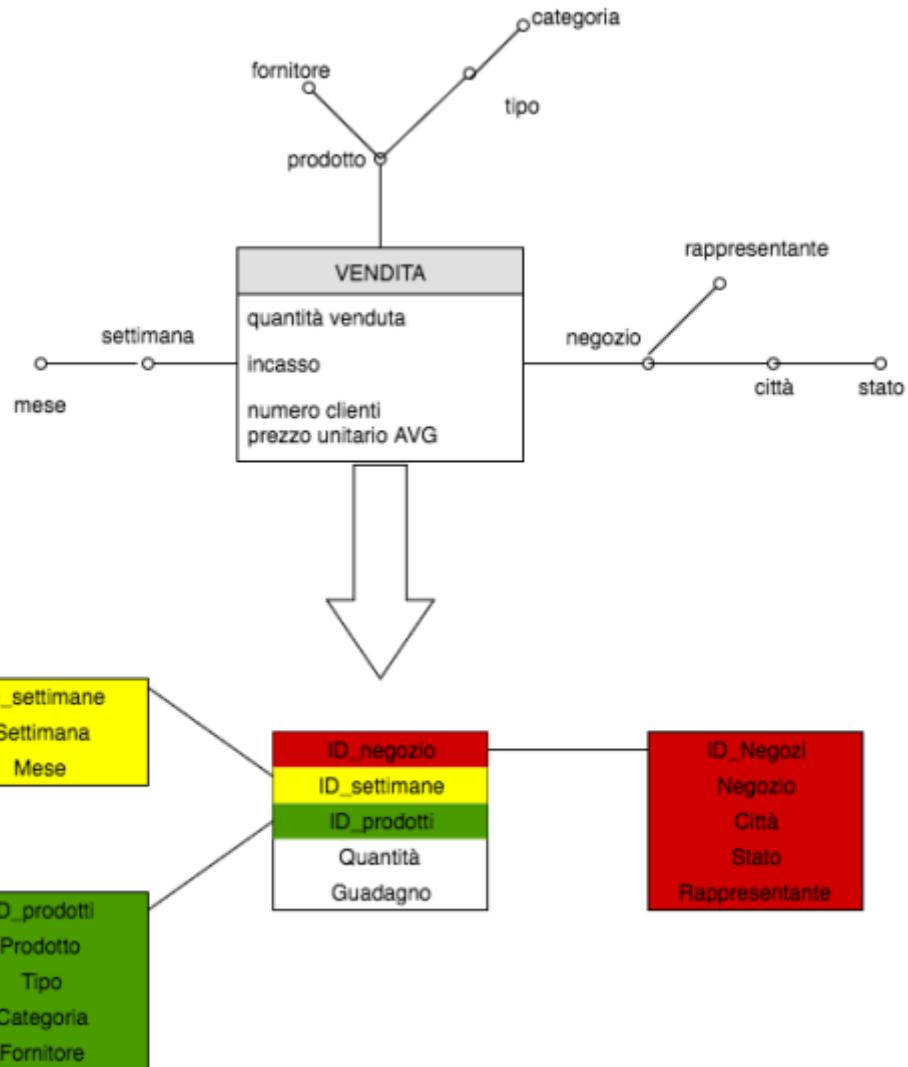


Figura 3.39: Star Schema del FACT Vendita

Vengono raccolti tutti i dati di tutti i negozi di un'azienda e così via, aggiungendo successivamente gli ID. La data è vista come un attributo: viene quindi “tirata fuori” e inserita nella gerarchia dimensionale. È importante osservare le dimensioni approssimative delle tabelle. Nel caso della Fact Table, considerando 1000 - 10000 scontrini al giorno saranno presenti circa 10^8 – 10^9 righe. La tabella prodotti invece, conterrà 1000 – 10000 righe che non crescono nel tempo: il catalogo dei prodotti resta circa costante. È presente quindi una differenza tra tabelle, suddivise in:

- Fact Table: tabelle grandi e in rapida crescita;
- Dimension Table: gerarchie relativamente piccole e statiche.

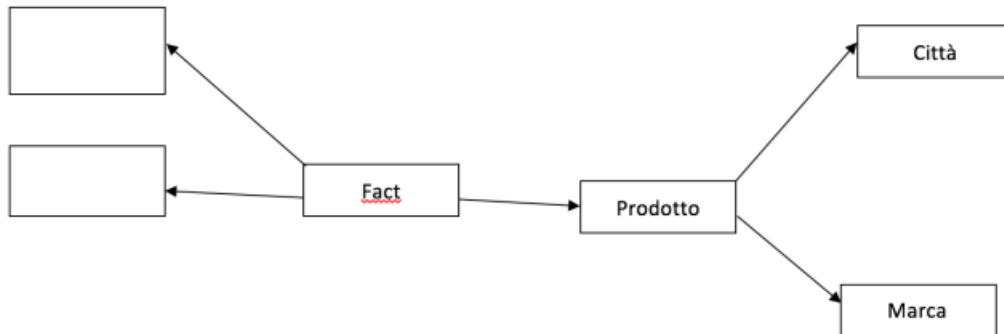
Si noti la differenza con il modello Entità – Relazioni, in cui le tabelle vengono trattate tutte allo stesso modo.

3.6.2 Motore Data Warehouse

Le tabelle le cui dimensioni variano poco nel tempo vengono create una sola volta, ad esempio quelle relative a dimensioni spaziali, dimensioni temporali, catalogo prodotti. La Fact Table, invece, è continuamente alimentata. A tal proposito si utilizza un sistema software come “Spoon”: questo è un tool grafico per rappresentare le trasformazioni sui pacchetti; questi vengono poi ripuliti sulla base della provenienza (Excel, Access, MySQL...) e infine inseriti. Infine un sistema come “Mondrian” crea gli indici effettuando i calcoli necessari.

3.6.3 SnowFlake Schema

Le Dimension Table sono completamente denormalizzate: è sufficiente una join per recuperare tutti i dati relativi a una dimensione. Tuttavia la denormalizzazione introduce una parte ridondante nei dati e devono quindi essere normalizzate; se sono presenti troppe join il sistema diventa complesso e difficile da gestire. Esiste la possibilità che esistano più fatti: nel caso di una università, essi possono essere gli esami, o ad esempio i voti degli assignment settimanali. Vengono quindi normalizzate anche altre tabelle, creando sottodimensioni ortogonali tra loro (e quindi indipendenti).



Ora la Fact Table punta ad una serie di dimensioni a loro volta strutturate con pezzi di gerarchie dimensionali. Lo SnowFlake schema può essere quindi visto come uno Star Schema ripetuto a più livelli. Per quanto riguarda le query, non ci sono differenze sostanziali anche se, a causa degli indici diversi, da un punto di vista fisico sono realizzate in maniera diversa. Si pensi ad una gerarchia condivisa con due fatti principali: ORDINE e SPEDIZIONE. Il primo di questi è aggregato in base a Magazzino ed Ordine: quest'ultimi possono essere raggruppati in base alla sottogerarchia spaziale Città (gerarchia di secondo livello).

Albero degli attributi: si tratta di diverse radici in corrispondenza di determinati fatti; sono usati per la creazione del modello concettuale.

Riconciliato: vengono riallineati i database relativi ai settori principali dell'azienda (ad esempio Marketing e Vendite).

L'ultima fase della progettazione di un Data Warehouse consiste nella progettazione del carico del lavoro. Infine i manager navigano gli ipercubi per ottenere i key performance indicator.

Ippazio Alessio
Simone Dongiovanni
01/12/2016

Appendici

3.7 Esercizio (Core di Facebook)

Analizziamo un esercizio riguardante il core di Facebook. Sul social network un autore ha la possibilità di scrivere dei Post ai quali possono essere aggiunti dei commenti o dei like, oppure possono essere condivisi da altri autori.

Le prime entità che modelliamo sono la Persona e Post.

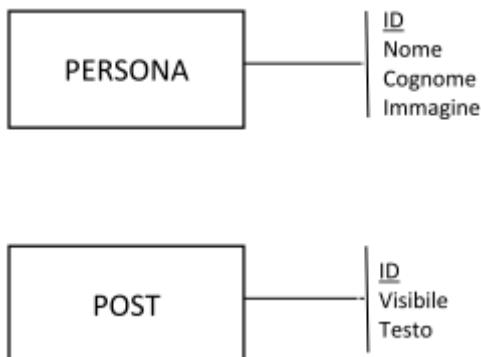


Figura 3.40: Entità Persona e Post

Cosa importante in Facebook sono i criteri di privacy per visualizzare i propri post. I criteri di privacy possono essere modellati come un tipo di entità oppure utilizzando il concetto di *metadatabase*. Esso è un database che definisce le regole di funzionamento di un altro database. In pratica viene definita una regola che viene scritta all'interno nell'entità “Criteri di Privacy” e permette di far vedere il post solo agli amici o agli amici degli amici (è un pezzo di database che ha un'influenza sul database senza avere una propria entità, è detto *meta-dato*).

Altre entità che andremo a modellare sono la Risorsa (rappresenta i riferimenti online che possono essere raggruppati in base al tipo), l'Album e il Gruppo.



Figura 3.41: Entità Risorsa ed Album



Figura 3.42: Entità Gruppo

Possono essere modellate due relazioni ricorsive su Persona e su Post:

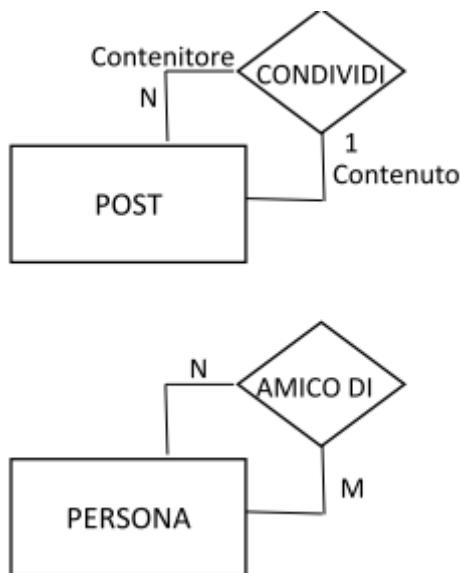


Figura 3.43: Relazioni: Post Condivide Post, Persona Amico Di Persona

Le relazioni che possono essere fatte tra Persona e Post sono:



Figura 3.44: Persona Scrive Post

Notiamo che fra Post e Scrive è presente il vicolo di partecipazione totale.

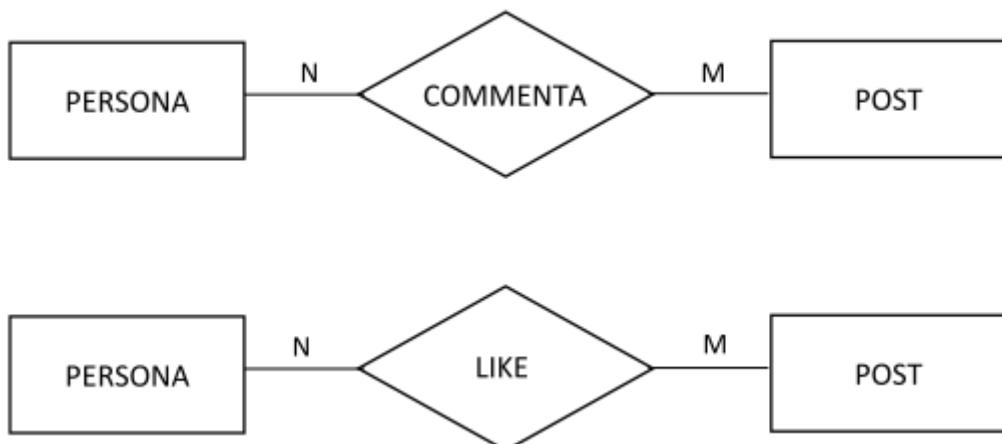


Figura 3.45: Persona Commenta Post, Persona Like Post

Invece tra Persona e Risorsa creiamo la relazione:

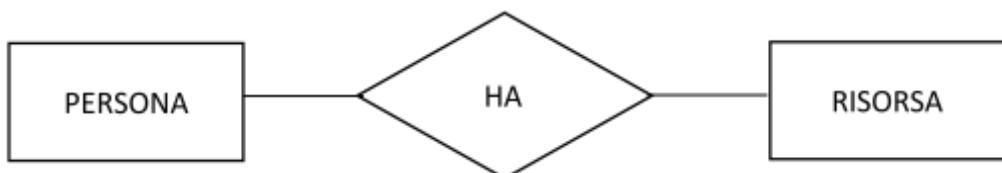


Figura 3.46: Persona Ha Risorsa

Tra Album e Risorsa è presente la relazione:



Figura 3.47: Risorsa Appartiene ad Album

Inoltre è presente una relazione ricorsiva su Album, chiamata “Contiene”

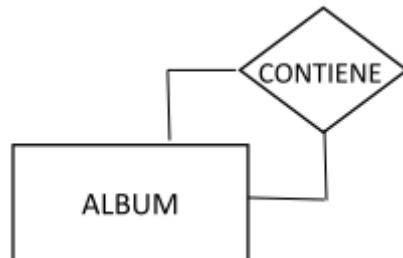


Figura 3.48: Album Contiene Album

L'entità Gruppo viene collegata alla Persona con la relazione “Crea” e con la relazione “Si iscrive” (utilizzo il pattern *Preventivo Consuntivo* per capire quando viene fatta la richiesta di iscrizione e quando viene accettato nel gruppo).

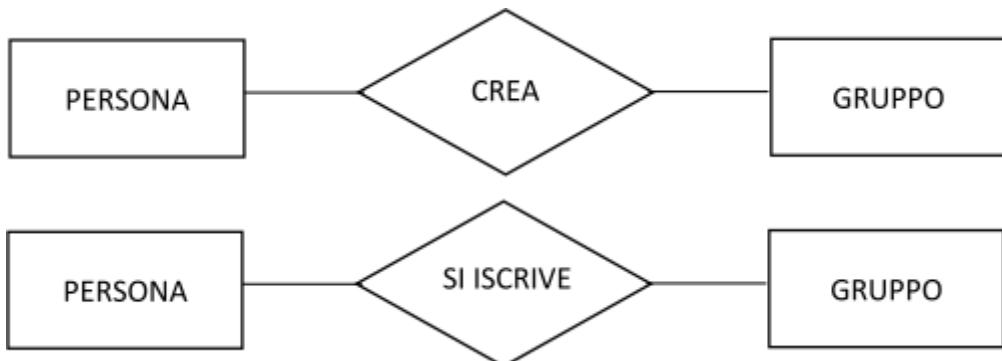


Figura 3.49: Persona Crea Gruppo, Persona Si Iscrive a Gruppo

Per ogni tipo di entità abbiamo almeno 2 tipi di vista, vista elenco (vengono distinte le persone online dalle persone generiche, o elenco solo degli amici) e vista di dettaglio. Lo schema finale del database sarà:

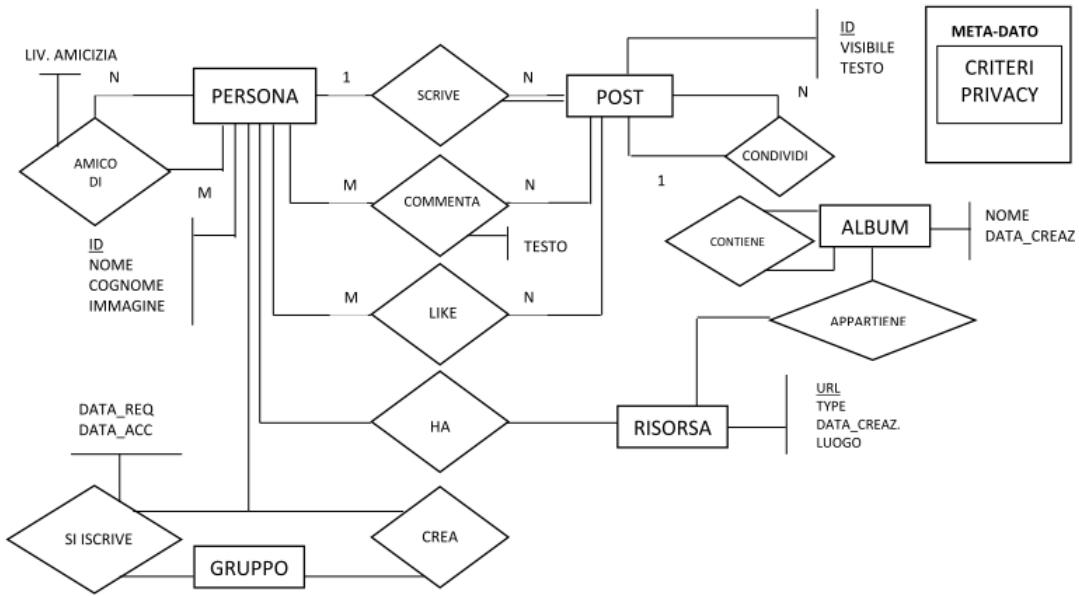


Figura 3.50: Core di Facebook - ER Finale

Gli stakeholders in questo esempio saranno:

- Amministratore;
- Utente;
- Amministratore dei gruppi.

3.7.1 Esempi di interrogazioni

- Dato un post fammi vedere quali commenti ha, chi ha scritto questi commenti, e quali altri commenti ha scritto uno stesso autore in altri post;
- Dato un post fammi vedere quali commenti ha, chi ha scritto questi commenti, e quali altri post ha scritto uno stesso autore;
- Dato un post condiviso, vedo il post originario e da chi è stato condiviso e la persona che è stato a condividerlo;
- Persona amico di persona che ha scritto post in ordine di data (Questa query rappresenta la bacheca di facebook).

```

1 SELECT POST.TESTO, COMMENTA.TESTO
2 FROM Persona AS p1, Amico_Di, Persona AS p2, POST, COMMENTA
3 WHERE p1.ID = Amico_Di.ID_Richiedente AND
4     p2.ID = Amico_Di.ID_Richiesto AND
5     POST.ID_Persona = P2.ID AND
6     COMMENTA.ID_POST = POST.ID AND
7     P1.Nome = "Signore" AND p1.Cognome = "X"

```

- Data una persona, chi è il suo “bersaglio” preferito?

```

1 SELECT p2.ID, p2.NOME, p2.COGNOME, COUNT
2 FROM PERSONA AS p1, COMMENTA_POST, PERSONA AS p2
3 WHERE p1.ID = COMMENTA.ID_PERSONA AND
4     COMMENTA.ID_POST = POST.ID AND
5     p2.ID = POST.ID_PERSONA AND
6     p1.NOME = "X" AND p2.COGNOME = "Y"
7 GROUP BY p2.ID

```

3.7.2 Meta-Database e Meta-Dati

Riprendiamo il concetto di Meta-Database e Meta-Dati. Per ottenere su MySQL l'elenco dei database, delle tabelle o delle viste basta effettuare attraverso delle query in un database particolare. All'interno del database esiste infatti l'Information Schema, nel quale sono contenute tutte le informazioni relative alle tabelle e agli attributi. Un meta-database non è altro che un database che descrive i dati di un altro oggetto (ad esempio di un altro database). Il data-catalog invece è l'insieme delle tabelle che descrivono i dati. È possibile realizzare anche un database per fare *introspezione* dei dati, cioè un DB che è in grado di analizzare la propria struttura. Il problema dell'introspezione è che non sappiamo come servircene, la utilizziamo entro certi limiti.

3.8 Modellazione

I modelli concettuali, logici e fisici vengono utilizzati per descrivere come realizzare delle applicazioni. Quando si realizza il diagramma entità relazioni in realtà si sta progettando anche il livello applicativo. Abbiamo utilizzato il modello Model-View-Controller che è molto vicino al concetto della 3-Layer-Architecture. I passi da effettuare per realizzare il modello quindi sono:

- Requirements:
 - Business Model Canvas (BMC) oppure E3Value (E^3V) (Si definisce contest e problema);
 - Goal & Stakeholder (Nell'esempio universitario gli Stakeholder sono lo Studente, il Professore e il Goal è la Laurea);
 - Requisiti (La soluzione del problema).
- Modellazione:
 - Hardware;
 - Rete;
 - Software:
 - * Presentation;
 - * Business Rule;
 - * Data (Enhanced Entity Relationship: in realtà è il primo passo da effettuare nella modellazione. Bisogna aggiungere delle annotazioni a questo diagramma riguardanti il dimensionamento della rete o dell'Hardware in base alle tabelle ottenute).
- Implementazione: In un DB posso usare il CRUD o le TRANSAZIONI ACID, senza usare programmazione a oggetti.

3.8.1 Stima della dimensione del database

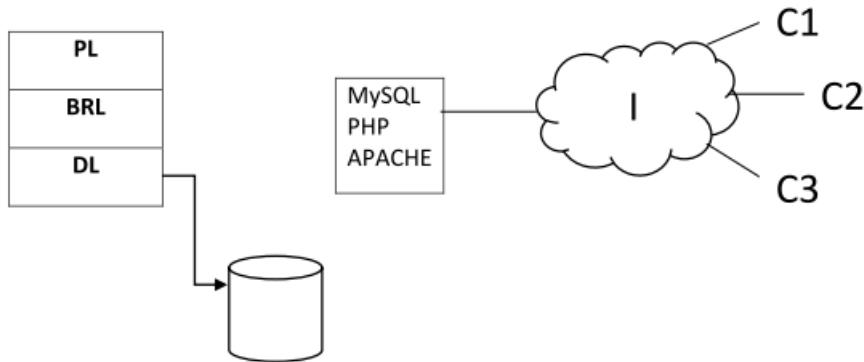


Figura 3.51: Architettura Logica DB

Il dimensionamento della rete e dell'hardware (Ram, Disco, Macchine Virtuali) si ottiene in base alle tabelle e alle relazioni.

Consideriamo ad esempio una semplice relazione:

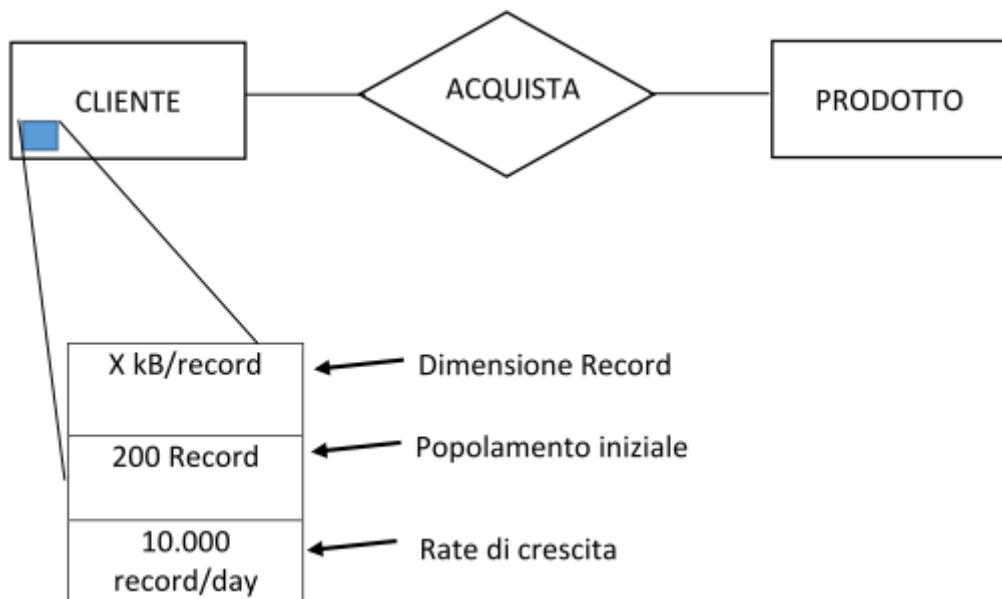


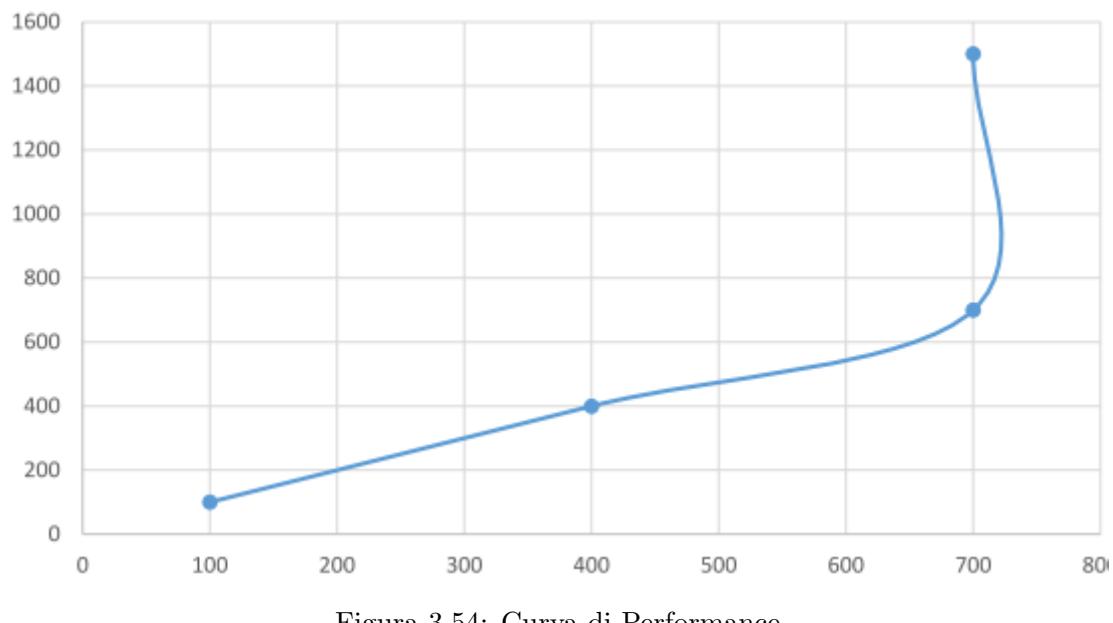
Figura 3.52: Cliente Acquista Prodotto - Dimensionamento

Tramite questi tre parametri indicati nella tabella il dimensionamento iniziale e la crescita successiva.

	INITIAL SIZE	TREND
A	2000 record * X kB/record	10k record/day * X kB/record
B
...
TOT	500 MB	2000 MB/Day

Figura 3.53: Cliente Acquista Prodotto - Dimensionamento tabellare

Tramite questa tabella riusciamo a capire che dopo un anno il nostro Hardware avrebbe bisogno di 2 Terabyte. Questa tecnica quindi ci permette anche di stimare la banda media per ottenere un certo tempo di risposta durante il giorno della nostra macchina.



3.8.2 MODELLO VISIBILITY

Per ogni Stakeholder presente è necessario disegnare il Visibility Model. Consideriamo nuovamente l'esempio precedente e realizziamo il visibility model per lo stakeholder 1 (S1):

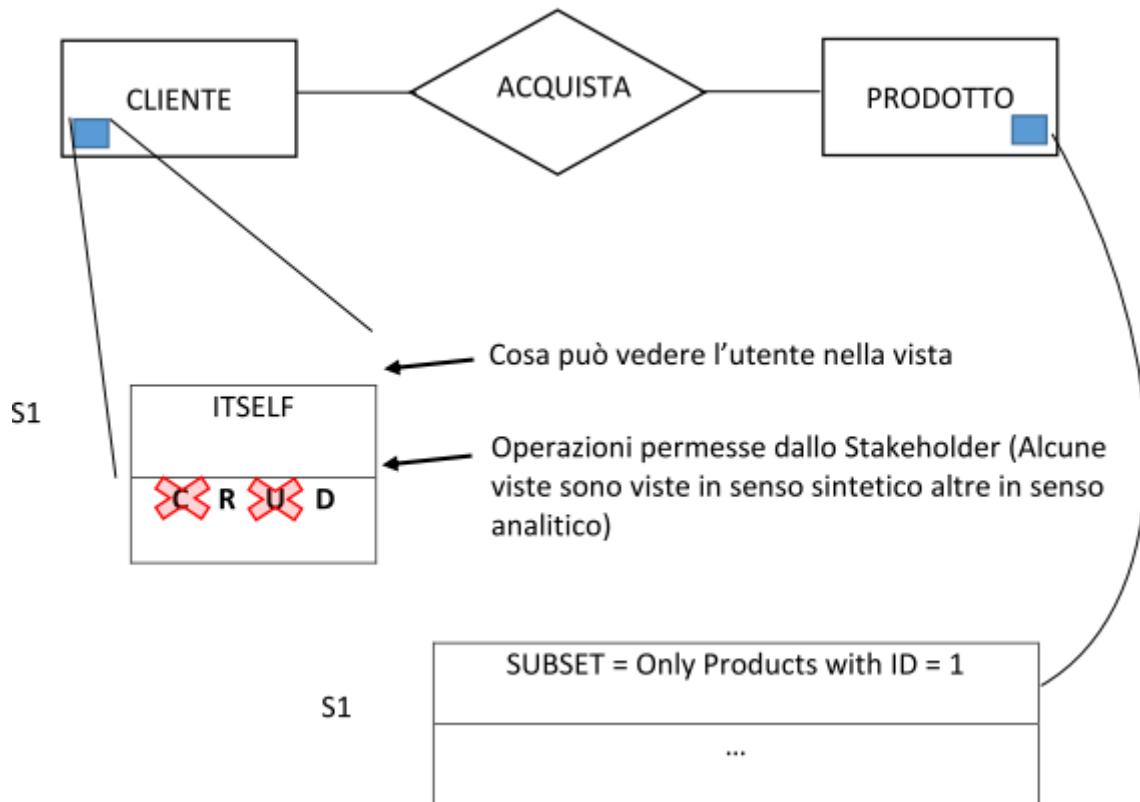


Figura 3.55: Modello Visibility di Cliente Acquista Prodotto

Anche in questo modello abbiamo due tipologie di navigazione:

- IN THE LARGE NAVIGATION: (Modello per mostrare le pagine e le connessioni fra di esse);
- IN THE SMALL NAVIGATION: (Modello per analizzare ogni aspetto singolarmente, come i Menù, le voci, i bottoni).

Angelo Cotardo
Francesco Filieri
13/03/2016

3.9 Esercizio (Compagnia Aerea)

Si analizzi il seguente caso di studio: Una nuova compagnia aerea vuole offrire un servizio di prenotazione e di acquisto di biglietti online, senza l'utilizzo alcuno di agenzie di viaggio né delle comuni biglietterie aeree ubicate presso gli aeroporti. Il sistema deve consentire le seguenti funzionalità:

- IN FASE DI PRENOTAZIONE VIA INTERNET:
 - Consultazione degli orari e delle destinazioni dei voli disponibili;
 - Consultazione di eventuali offerte promozionali;

- Scelta dell’orario e della destinazione;
- Inserimento dei dati anagrafici dei passeggeri e delle eventuali preferenze (pasto, fumatori, ecc...);
- Scelta del posto preassegnato (tenendo conto delle precedenti prenotazioni e delle preferenze);
- Calcolo della tariffa da applicare (tenendo conto di eventuali offerte promozionali);
- Addebito, previa conferma, del costo del biglietto sulla carta di credito fornita (sicurezza tramite protocollo SSL);
- Stampa del biglietto elettronico che riporta il codice del biglietto i dati del/i passeggero/i, della partenza, della destinazione e il costo dettagliato (tariffa del biglietto, tasse, etc.).

• IN FASE DI CHECK-IN PRESSO L’AEREOPORTO DI PARTENZA:

- Variazione del posto preassegnato (a seconda della disponibilità dei posti e delle precedenti prenotazioni e check-in);
- Inserimento della segnalazione dei bagagli imbarcati;
- Emissione della carta d’imbarco.

• IN FASE DI CONSUNTIVAZIONE VOLI:

- Riscontro delle carte d’imbarco emesse;
- Emissione di riepiloghi sui voli effettuati e sui passeggeri;
- Analisi delle preferenze dei passeggeri ed invio di mailing mirate per informare di nuove promozioni e di servizi specifici per la clientela (ad esempio per viaggiatori che viaggiano spesso per vacanze in luoghi esotici, mailing per informarli di nuove destinazioni e di nuove offerte sulle loro rotte abituali).

Sulla base della seguente traccia, viene realizzato quello che è il modello E/R:

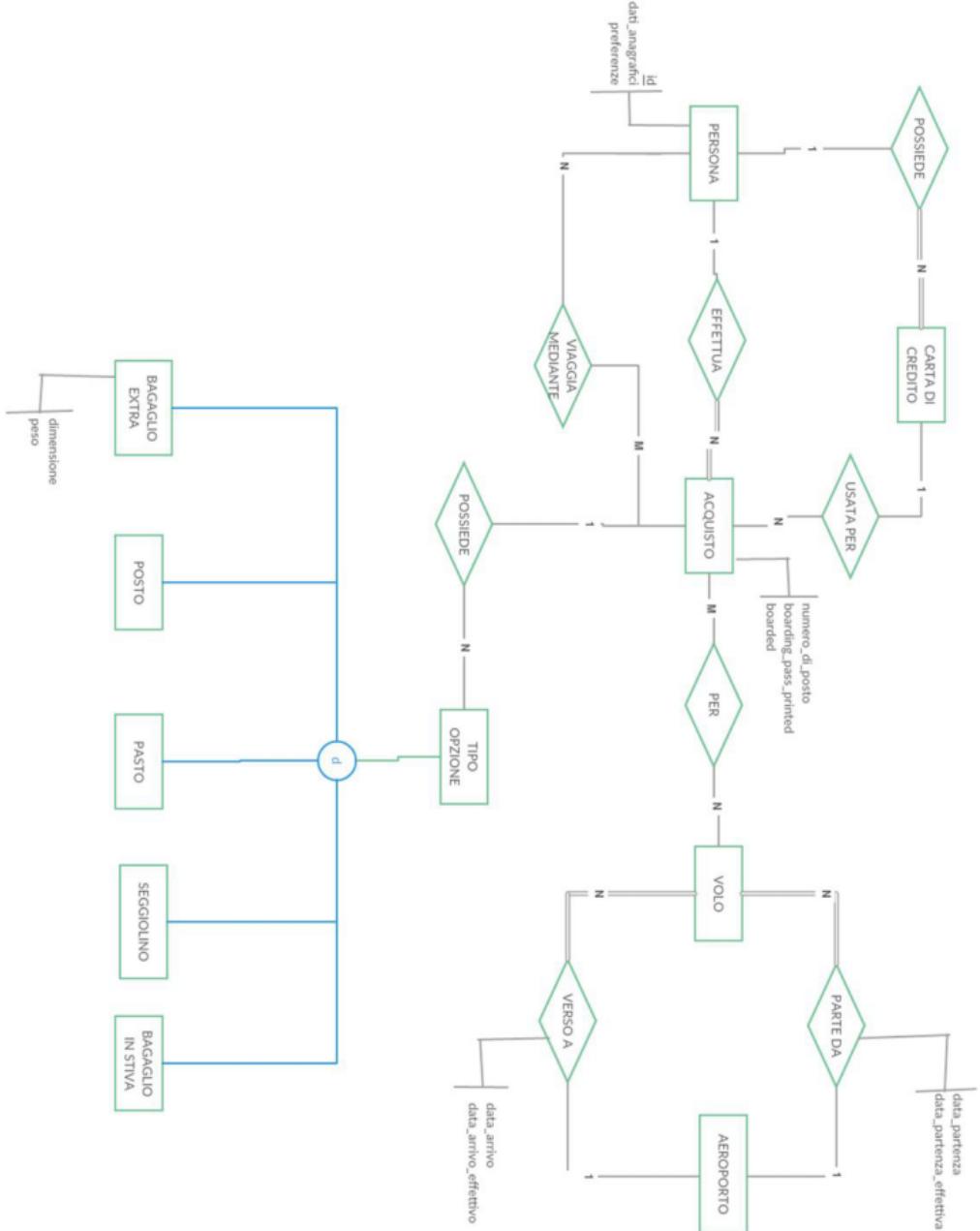


Figura 3.56: Compagnia aerea - ER Finale

Nel diagramma riportato sopra si ha una possibile modellazione del caso di studio analizzato. La fase di modellazione è iniziata facendo riferimento all'azione principale che si deve compiere, cioè l'acquisto del biglietto aereo da parte del cliente.

Conviene non usare il termine biglietto, ma conviene scegliere un nome differente per evitare di creare problemi nel medio/lungo termine.

Perciò conviene usare un'entità che tenga traccia della transazione di acquisto, ed ecco perché si userà, come vedremo, l'entità *Acquisto*.

Questo porta alla creazione di una relazione in cui le entità in gioco, come si può notare osservando la figura, sono:

- *Persona*;
- *Acquisto*.

Dato che una *Persona* deve effettuare un *Acquisto*, tenendo conto che gli acquisti possono essere eseguiti unicamente online, questa deve avere una o più *Carte Di Credito* associate al suo profilo. È errato considerare la *Carta Di Credito* come attributo di *Persona* perché ad ogni individuo può essere associata più di una carta oppure potrebbe esserne sprovvisto, il che porterebbe ad avere un eccessivo numero di NULL nel database. Inoltre c'è la **PARTECIPAZIONE TOTALE** dal lato della *Carta Di Credito*, in questo modo ogni carta deve avere un titolare. Nel nostro modello E/R, non intendiamo l'entità *Persona* solo come acquirente, ma viene intesa anche come passeggero, perché un utente può anche acquistare un biglietto per terzi. In fase di *Acquisto* è possibile selezionare delle opzioni, per tanto si è deciso di specializzare il *Tipo Di Opzione* in modo da associare all'*Acquisto* le varie proposte che la compagnia offre al cliente. Ad esempio:

- *la selezione del posto*;
- *l'aggiunta di un bagaglio extra con relative dimensioni e peso*;
- *la selezione del tipo di pasto da consumare a bordo*;
- *la presenza del bagaglio da portare in stiva*;
- *la richiesta del seggiolino per bambini*.

L'entità *Acquisto* ha come attributi necessari al soddisfacimento dei requisiti:

- *numero di posto*, che è utile per tenere traccia del numero di biglietti già venduti e per far selezionare al cliente il posto desiderato o assegnarne uno casuale;
- *boarded_pass_printed*, che è utile alla compagnia aerea per capire quanti passeggeri hanno stampato la carta d'imbarco e quindi il numero di persone che hanno intenzione di usufruire del volo;
- *boarded*, che indica se la persona è effettivamente a bordo o meno.

Come si intuisce facilmente, gli ultimi due attributi dell'entità *Acquisto* saranno di tipo booleano. È stato scelto, inoltre, di modellare l'entità *Volo*. Tale entità è in doppia relazione con l'entità *Aeroporto*, in quanto, ciò ci permette di tenere traccia dell'*Aeroporto* di partenza e quello di destinazione. Le due relazioni hanno come attributi:

- *data prevista*;

- *data effettiva.*

il che consente di monitorare eventuali ritardi e di avere un calendario completo dei voli. In una prima analisi si era deciso di creare un'entità Giorno, che era collegata in una relazione ternaria con le entità volo e aeroporto e le relazioni “parte da” - “verso”, allo scopo di creare un calendario dei voli. Questa procedura ha senso solo nel momento in cui l'entità giorno possiede diversi attributi che la caratterizzano o se il giorno deve contenere informazioni importanti (quindi si deve distinguere dagli altri giorni, come giorni da bollino nero, sconti per i festivi, ...), in modo da fornirle una certa dignità informativa, altrimenti se contiene semplicemente l'attributo data, allora la si considera come un attributo di una determinata entità o relazione. Infine c'è da gestire la questione degli scali, per poterlo fare si “gioca” con la cardinalità della relazione “per” che collega gli attributi *Acquisto* e *Volo*.

Cristian Annicchiarico
 Mattia Marzano
 01/12/2016

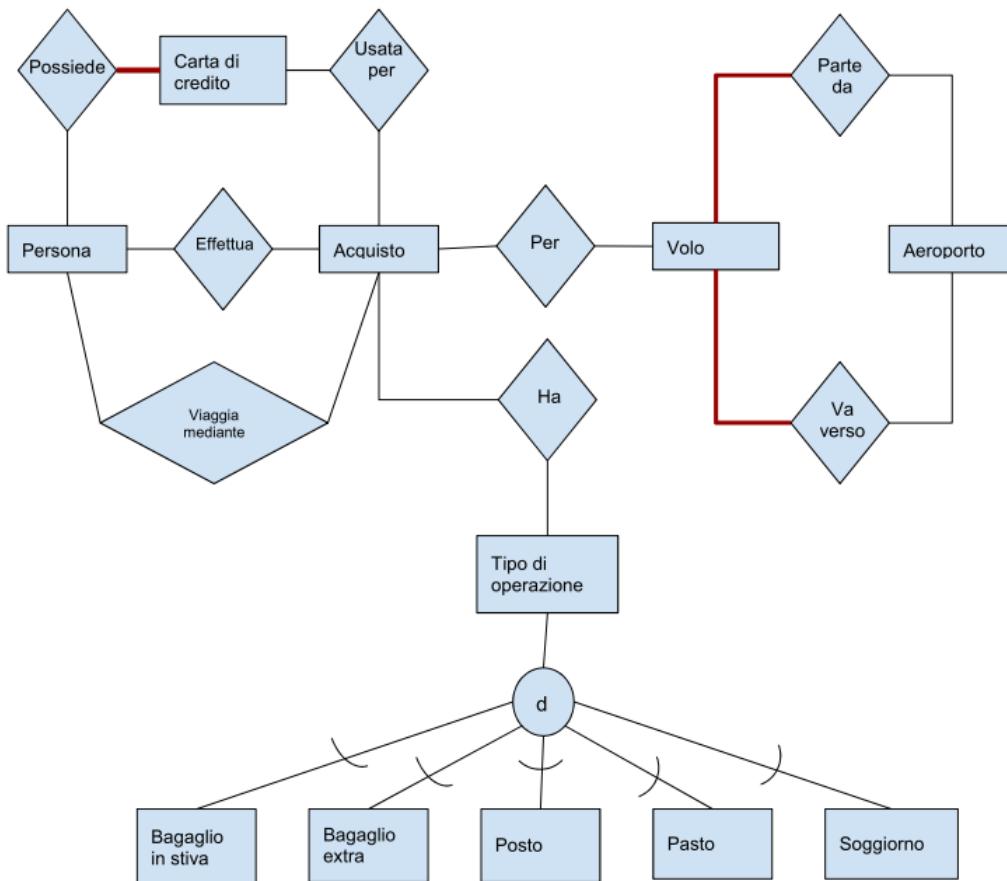


Figura 3.57: Compagnia aerea - ER Finale - RECAP

3.9.1 Continuazione ESERCITAZIONE

Riprendiamo in esame lo scenario analizzato nella precedente lezione riguardante la gestione dei voli da parte di una compagnia. Un'analisi più dettagliata ci ha permesso di cogliere

delle problematiche che in prima approssimazione del modello relazionale non è stato possibile osservare. Il problema di questo schema si presenta principalmente con l'analisi della situazione in cui l'acquisto dei biglietti è multiplo. Qualora venisse effettuato un acquisto multiplo di biglietti, risulterebbe difficile l'implementazione dell'inserimento di un'opzione aggiuntiva da parte di un singolo cliente appartenente ad un gruppo, in modo indipendente. Analizziamo più nel dettaglio cosa intendiamo per gruppo. Per gruppo in questo caso stiamo intendendo un agglomerato di persone. Vogliamo poter tenere traccia dei dati di ogni singolo passeggero, in quanto potrebbero essere potenziali clienti futuri per la compagnia di voli. Il problema fondamentale con il “gruppo” è: una persona che viaggia in gruppo può comprare un'opzione personale? Cerchiamo di modificare lo schema riportato sopra per risolvere il nostro problema.

Ci possono essere almeno due modi generalmente validi per risolvere il problema:

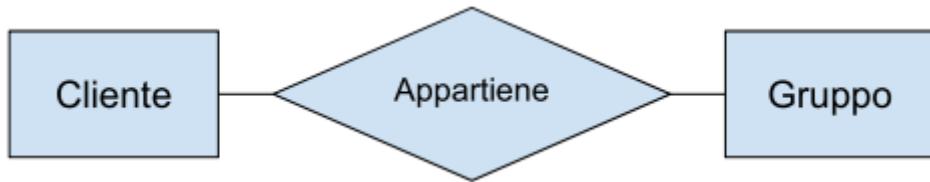


Figura 3.58: Cliente Appartiene a Gruppo

Se consideriamo il primo schema per modellare un acquisto che si riferisce ad un gruppo di persone nel caso in cui un acquisto venisse effettuato da un cliente per una sola persona potremmo pensare che la tabella Gruppo presenti degli attributi a NULL. Riflettendo attentamente lo schema in figura 1 non vieta che il gruppo sia costituito da una sola persona pertanto non assistiamo alla presenza di NULL nella tabella Gruppo quanto ad una replicazione degli attributi di Cliente. In generale una schematizzazione siffatta è preferibile quando due oggetti sono eterogenei;

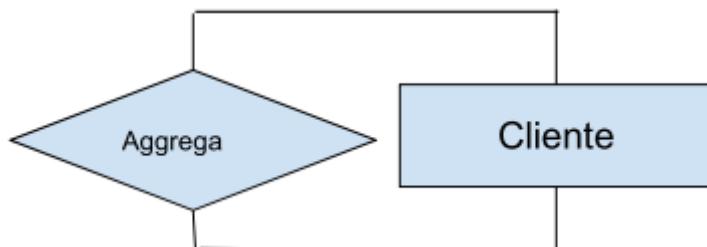


Figura 3.59: Cliente Aggrega Clienti

Con questo schema mettiamo in relazione l'entità Cliente con se stessa attraverso la relazione Aggrega. Notiamo che per i nostri propositi questa schematizzazione non va bene dal momento che Cliente e Gruppo non sono oggetti omogenei. In generale questo tipo di schematizzazione è preferibile quando gli oggetti sono omogenei; un esempio può essere una gerarchia di Persone come in un albero genealogico.

Query SQL

- Numero di CC della persona con nome Mario Rossi:

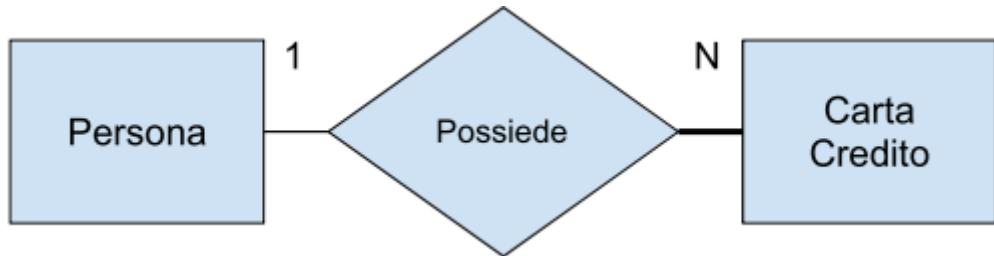


Figura 3.60: Persona Possiede Carta di Credito

```

1 SELECT COUNT(*)
2 FROM PERSONA JOIN CARTACREDITO ON COD.PERSONA = ID.PERSONA
3 WHERE PERSONA.NOME = "Mario Rossi"

```

- Tutti i voli dell'aeroporto con nome “Aeroporto1”:

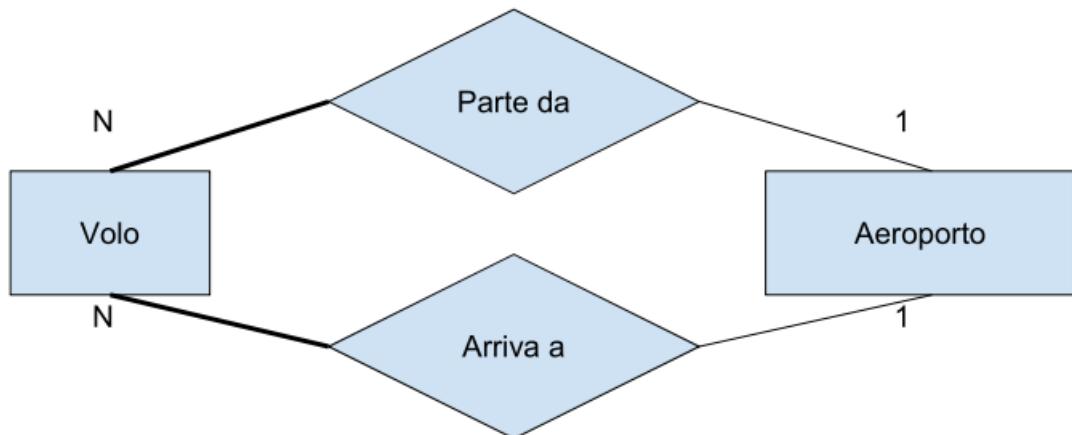


Figura 3.61: Volo Parte Da/Arriva A Aeroporto

```

1 SELECT *
2 FROM VOLO
3 JOIN AEROPORTO ON VOLO.ID_AEROPORTO_PARTENZA = AEROPORTO.
     COD_AEROPORTO
4 WHERE AEROPORTO.NOME = "Aeroporto1"

```

- Estrazione della coppia di opzioni più acquistate (insieme):

```

1 SELECT MAX(CONTEGGIO) , TO1.COD_OPZIONE, TO2.COD_OPZIONE
2 FROM (SELECT TO1.COD_OPZIONE, TO2.COD_OPZIONE, COUNT(*) AS CONTEGGIO
3 FROM TIPO_OPZIONE AS TO1, TIPO_OPZIONE AS TO2
4 WHERE TO1.COD_ACQUISTO = TO2.COD_ACQUISTO)

```

OSSERVAZIONE: Sebbene questa query sia più articolata rispetto alle precedenti non si tratta di una subquery dal momento che la query annidata si trova nello statement FROM e non nel WHERE

- Elenco di tutte le opzioni e del numero di volte che sono state vendute nell'anno 2016:

```

1 SELECT COUNT(*) , TO.COUNT(*)
2 FROM TIPO_OPZIONE AS TO
3 JOIN INCLUDE AS I ON TO.COD_OPZIONE = I.ID_OPZIONE
4 WHERE YEAR(INCLUDE.DATA) = 2016
5 GROUP BY TO.COD
```

OSSERVAZIONE: Nello statement WHERE la condizione sulla data poteva essere scritta in termini di INCLUDE.DATA LIKE “_ _/_ _/2016”. Questa espressione non è sbagliata ma introduce operazioni di pattern matching che degradano le prestazioni della query.

3.9.2 DFM

Partiamo dal Database OLTP transazionale, aggiungendo gerarchie spaziali, temporali e domain specific si costruisce il Database Multidimensionale e successivamente si individuano i fatti principali.

FATTI PRINCIPALI

La determinazione dei fatti principali consiste nell'individuazione nel database riconciliato di quelle entità che sono interessate da un numero molto grande di transazioni. Nel nostro caso:

- Acquisto;
- In;
- Richiede;
- Scalo

Prendiamo in considerazione il fatto “Scalo” i cui attributi sono Orario effettivo, Orario previsto e Tipo:

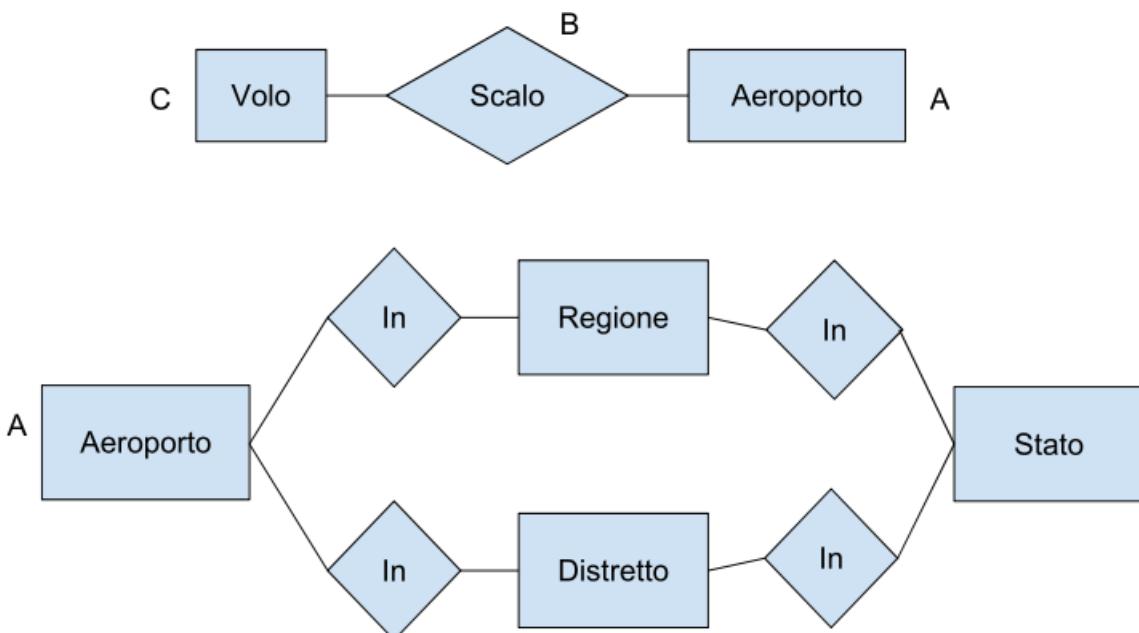


Figura 3.62: Scalo FACT

- A: Da Aeroporto scaturisce una gerarchia spaziale per Scalo;
- B: Dagli attributi Orario effettivo e Orario previsto scaturiscono gerarchie temporali:

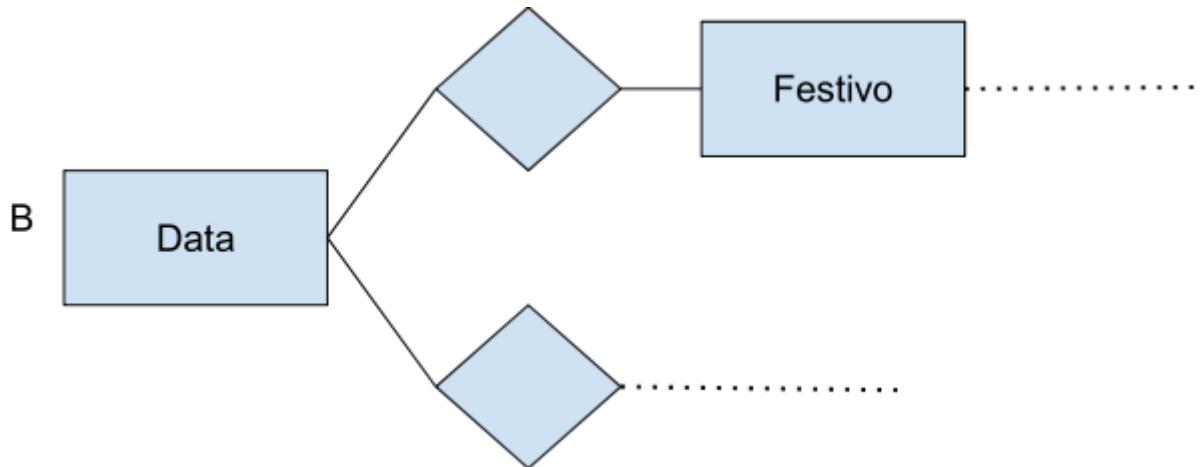


Figura 3.63: Gerarchia ER - Data

- C: Anche dall'entità Volo scaturisce una gerarchia relativa al fatto Scalo:

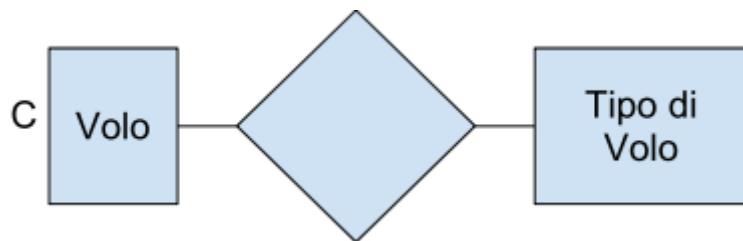


Figura 3.64: Gerarchia ER - Volo

FACT MODEL

Potrebbe essere interessante analizzare il fatto Decollo. Si noti che le misure rispetto cui il fatto è analizzato sono attributi calcolati, come ad esempio il numero dei passeggeri.

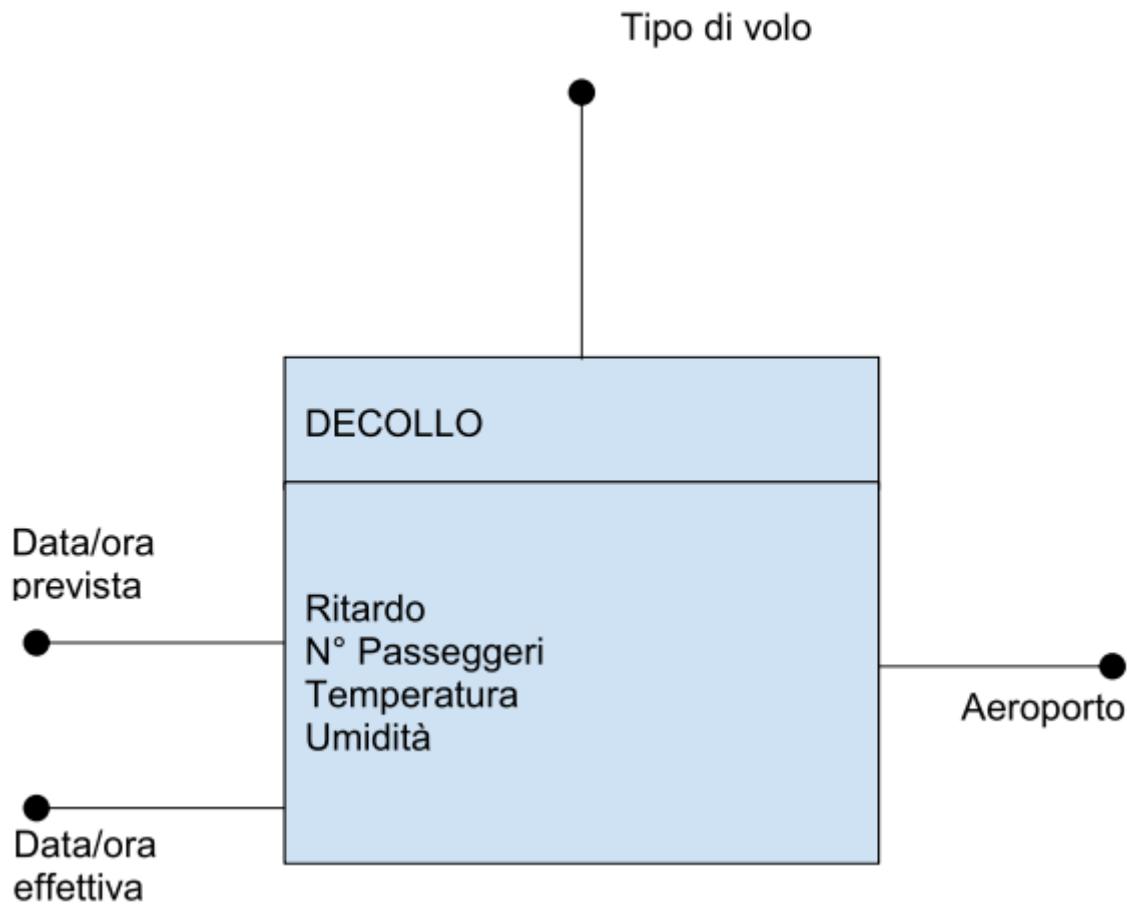


Figura 3.65: Decollo FACT

Gabriele Accarino
 Emanuele Costa Cesari
 07/12/2016

3.10 DATABASE TECHNOLOGIES

3.10.1 CLOUD COMPUTING

Il cloud computing è l'utilizzo di risorse digitali distribuite in maniera elastica usufruendo di Internet e serve per inquadrare big data e database NoSQL (Not only SQL). Quando parliamo di cloud presupponiamo di accedere a questo tramite protocolli HTTP o TCP-IP (dove per HTTP intendiamo il livello applicativo che poggia sul servizio di rete TCP-IP).

Poiché col cloud computing la struttura non è fissa, la quantità di risorse utilizzate cambia dinamicamente: dal punto di vista del provider lo spazio viene redistribuito a seconda dei parametri economici e di servizio mentre da quello del client viene effettuata una richiesta a seconda della necessità.

Il cloud computing ha così rilassato il concetto di dimensionamento ed allocazione delle risorse: quando andiamo a progettare un sistema non abbiamo più il bisogno di vedere preventivamente di quanto spazio, potere computazionale o banda necessita il nostro data center e comprarlo a priori. Infatti, i sistemi informatici, fino all'arrivo del cloud computing si sono

basati sul concetto di massimo carico per il dimensionamento cercando poi di spalmare il costo delle risorse negli anni.

Oggi, quindi, col cloud computing non ragioniamo più dimensionando sul picco, ma riallocando la memoria in base alla richiesta media e contenendo gli eventuali picchi. Possiamo definire dei parametri di servizio che dinamicamente possono fare aumentare la potenza di elaborazione del server in modo che il data center sia in grado di gestire eventuali situazioni di carico.

Dietro a questo nuovo scenario tecnologico c'è l'attuale modello di business (e.g. Gmail e servizi correlati). Abbiamo una serie di servizi gratuiti in cambio dei nostri dati che modellano la pubblicità: Google o qualsiasi altro provider vende le nostre info personali aggregate.

Il cloud computing prevede tre tipologie: IaaS (Internet as a Service), PaaS (Platform as a Service), SaaS (Software as a Service). Con questi possiamo comprare storage e capacità computazionale: non diventiamo possessori ma solo utilizzatori di questi oggetti, che sono regolati da un contratto di servizio. Questi contratti sono definiti SLA (service level agreement) e determinano la minima qualità del servizio erogato. Esistono inoltre paradigmi sul cloud computing: ad esempio il NIST americano dove vengono rappresentati i vari attori e le varie funzioni all'interno dell'architettura cloud.

Un problema fondamentale è quello del posizionamento fisico dei dati per garantire la protezione della privacy degli stessi. La privacy, per legge, copre info su salute, minori, religione e sesso, in tal modo possiamo determinare se un dato è riservato o è protetto da privacy e, quindi, anche il costo del dato stesso. Data protection o privacy sono elementi fondamentali quando si parla di cloud computing. Attualmente tutti i grossi provider di servizi stanno creando grossi data center cercando di portare sul cloud tutti i servizi critici commodity (servizi richiesti ma di pari qualità da parte di tutti i provider): per far ciò fanno uso di database distribuiti. Un database distribuito prevede che i dati possano essere distribuiti geograficamente su più luoghi.

Dobbiamo ad ogni modo avere un modello dati consistente per gestire come questi dati sono deployati tra i vari nodi. Ci sono tecniche per ottimizzare efficienza e gestione: un esempio può essere quello di considerare il database distribuito come centralizzato. Tuttavia, quando ci troviamo in un'ambiente distribuito di dati non possiamo verificare contemporaneamente tutte le prerogative del mondo ACID (secondo la teoria di Brewer).

Riportandoci al concetto di database di tipo transazionale, basato su modelli relazionali, le transazioni devono essere ACID (Atomicity, Consistency, Isolation, e Durability). Il limite delle relazioni è che funzionano benissimo quando parliamo di dati semplici, ma se facciamo uso di file multimediali non vanno più bene le query: da qui è nato il mondo dei database Not only SQL: sono quei database che non hanno necessità di rispondere alle caratteristiche delle transazioni ACID.

Un'ultima, ma non meno importante, caratteristica del cloud computing è il multi-tenance: data una infrastruttura permettiamo che ci siano diversi proprietari di dati. Questi però al momento della creazione del database possono decidere quali dati sono isolati e quali condivisi.

3.10.2 NIST REFERENCE ARCHITECTURE

Nel cloud computing abbiamo diversi attori:

- Cloud Consumer: il principale stakeholder del servizio;
- Cloud Provider: l'ente responsabile della disponibilità del servizio;
- Cloud auditor: chi fa il monitoraggio dei servizi, performance, gestisce clausole degli SLA;

- Cloud broker: chi effettua dei servizi di intermediazione tra domanda e offerta. Se gli SLA del servizio non sono garantiti, il broker avverte il provider del servizio.

NIST Reference Architecture

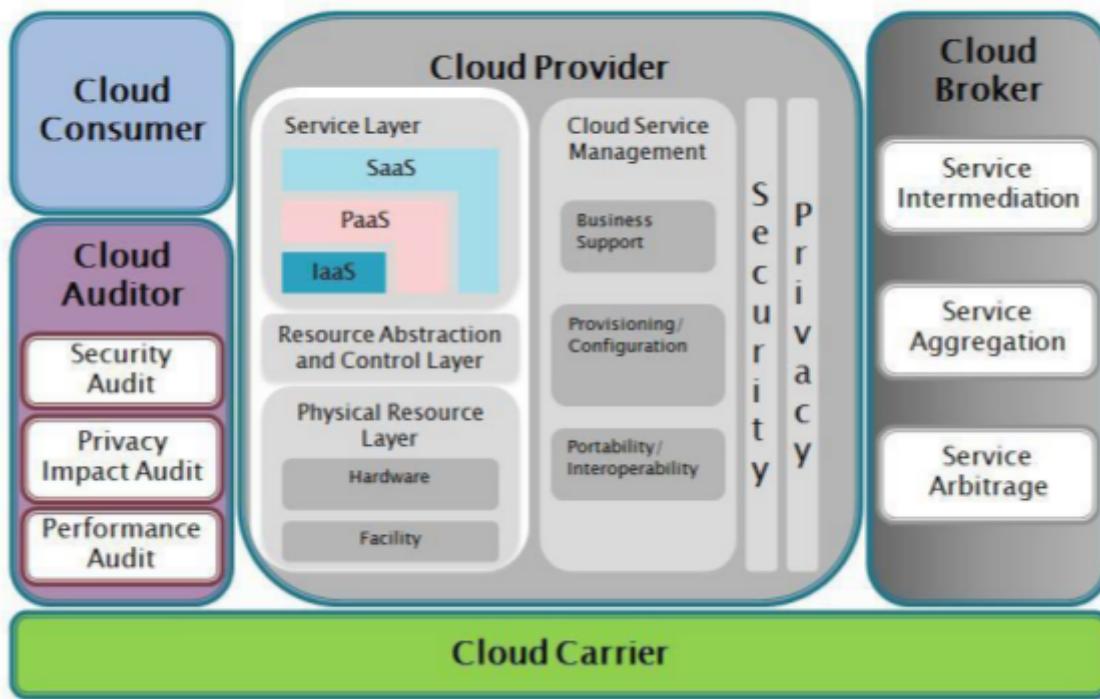


Figura 3.66: Architettura NIST

3.10.3 BIG DATA

La crescita è stata esponenziale dalla nascita di internet: parliamo di “valanga di informazioni” o peta bytes di dati che vengono gestiti. Quando ci riferiamo ai big data non badiamo solo al volume di dati ma anche ad altre caratteristiche particolari quali veridicità e velocità delle informazioni.

Il famoso CRUD (Create, Read, Update, Delete) del database si applica a questa grossa mole di dati che si può presentare in varie forme (infatti il tipo di visualizzazione è un tema abbastanza importante). È necessario quindi un sistema di “early warning” per fare l’analisi.

3.10.4 CAP'S THEOREM

Il teorema CAP, noto anche come teorema di Brewer, afferma che è impossibile per un sistema informatico distribuito fornire simultaneamente tutte e tre le seguenti garanzie:

- Coerenza: (tutti i nodi vedono gli stessi dati nello stesso momento);
- Disponibilità: (la garanzia che ogni richiesta riceva una risposta su ciò che è riuscito o fallito);

- Tolleranza di partizione: (il sistema continua a funzionare nonostante arbitrarie perdite di messaggi).

Secondo il teorema, un sistema distribuito è in grado di soddisfare al massimo due di queste garanzie allo stesso tempo, ma non tutte e tre.

3.10.5 DATABASE NoSQL

I database NoSQL (not only SQL) forniscono metodi di immagazzinamento e ricerca dati che sono modellati diversamente rispetto ai database relazionali, alcuni esempi sono:

- Key-value (k-v): formato da coppie chiave valore;
- Columnar: formato perlopiù da colonne anziché righe;
- Document: per gestire grosse moli di dati testuali (e.g., MongoDB o CouchDB);
- Graph: formato da dati strutturati a grafo (e.g. Neo4J).

3.10.6 MAP REDUCE

MapReduce è un modello di programmazione per processare grandi set di dati. Si rifà alla struttura delle funzioni map e reduce utilizzate in programmazione funzionale. MapReduce è un framework per l'elaborazione di problemi parallelizzabili attraverso grandi dataset di dati che utilizzano un vasto numero di computer (nodi). L'elaborazione computazionale dei dati può essere o in un file system (non strutturato) o in un database (strutturato). Apache Hadoop è stato il primo sistema di map reduce in parallelo. Spark ha successivamente sostituito Hadoop.

- Step "Map": Il nodo master prende l'input, lo divide in piccoli sotto-problemi, e distribuisce questi ai nodi computazionali. Questi ultimi possono ripetere a loro volta tale procedura, portando così ad una struttura ad albero multilivello. A questo punto il nodo operativo elabora il problema più piccolo e poi passa la risposta al suo nodo master;
- Step "Reduce": Il nodo master raccoglie le risposte a tutti i sotto-problemi e li combina in qualche modo per formare la risposta al problema che sta cercando di risolvere.

Federico De Luca
Giuseppe D'Amuri
15/12/2016

Thanksgiving

Si ringrazia:

- Naturalmente in primis alla nostra fantastica classe del II anno di Ingegneria Informatica at UNISALENTO, anno 2016/2017;
- il prof. *Mario Alessandro Bochicchio* per il corso di Database tenutosi in AA 2016/2017 del corso di laurea in Ingegneria Informatica at UNISALENTO, ed anche per disponibilità e chiarimenti. Contatto: *mario.bochicchio@unisalento.it*;
- me stesso, *Marco Chiarelli*, studente del II anno di Ingegneria Informatica at UNISALENTO. Contatti: *{marco_chiarelli@yahoo.it, marcochiarelli.nextgenlab@gmail.com, marco.chiarelli@studenti.unisalento.it}*;
- La mia squadra di studio: *{Gabriele Accarino, Matteo Settembrini, Paolo Panarese, Emanuele Costa Cesari, Dino Sbarro}*;
- Google, ovviamente.

Credits



/marcochiarelli

Bibliografia

- [1] R. Elmasri, S. Navathe, *Fundamentals of Database Systems, seventh edition*, Pearson International
- [2] M. Golfarelli, S. Rizzi *Datawarehouse Design-Modern Principles and Methodologies*, McGraw-Hill