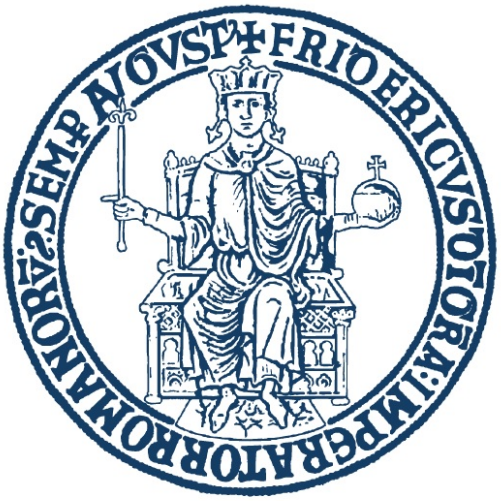


Università degli Studi di Napoli Federico II

Università degli Studi di Napoli Federico II

Corso di Laurea Triennale in Informatica

Documentazione Hackathon

Basi Di Dati I

Mario Majorano	N86005035
Luca Sanselmo	N86005147

Anno Accademico 2024/2025

Indice

Introduzione

La presente documentazione ha l'obiettivo di:

1. Esporre l'analisi dei requisiti condotta a partire dal problema proposto dal cliente
2. Illustrare la progettazione concettuale sviluppata sulla base dell'analisi effettuata
3. Presentare lo schema logico del modello effettuata al fine di permetterne l'implementazione
4. Descrivere la base di dati concreta, traducendo gli elementi determinati nelle fasi precedenti in tabelle fisiche

1 Specifica e Analisi dei Requisiti

1.1 Definizione del Dominio

I requisiti sono specificati all'interno della seguente traccia di esercizio:

Traccia del Problema

Un hackathon, ovvero una "maratona di hacking", è un evento durante il quale team di partecipanti si sfidano per progettare e implementare nuove soluzioni basate su una certa tecnologia o mirate a un certo ambito applicativo.
Ogni hackathon ha un titolo identificativo, si svolge in una certa sede e in un certo intervallo di tempo (solitamente 2 giorni) e ha un organizzatore specifico (registrato alla piattaforma). L'organizzatore seleziona un gruppo di giudici (selezionati tra gli utenti della piattaforma, invitandoli). Infine, l'organizzatore apre le registrazioni, che si chiuderanno 2 giorni prima dell'evento. Ogni evento avrà un numero massimo di iscritti e una dimensione massima del team.
Durante il periodo di registrazione, gli utenti possono registrarsi per l'Hackathon di loro scelta (eventualmente registrandosi sulla piattaforma se non lo hanno già fatto). Una volta iscritti, gli utenti possono formare team. I team diventano definitivi quando si chiudono le iscrizioni. All'inizio dell'hackathon, i giudici pubblicano una descrizione del problema da affrontare.
Durante l'hackathon, i team lavorano separatamente per risolvere il problema e devono caricare periodicamente gli aggiornamenti sui "progressi" sulla piattaforma come documento, che può essere esaminato e commentato dai giudici. Alla fine dell'hackathon, ogni giudice assegna un voto (da 0 a 10) a ciascun team e la piattaforma, dopo aver acquisito tutti i voti, pubblica le classifiche dei team.

1.2 Obiettivi del Sistema

A partire dalla traccia fornita, il sistema da progettare deve consentire:

- La gestione completa del ciclo di vita di un hackathon
- La registrazione e autenticazione degli utenti sulla piattaforma
- La formazione e gestione dei team di partecipanti
- Il caricamento e la valutazione dei documenti prodotti
- La generazione automatica delle classifiche finali

1.3 Analisi dei Requisiti

1.3.1 Requisiti Informativi

Il sistema prevede la realizzazione di una piattaforma atta all'organizzazione e alla gestione di una maratona di hacking in cui team di partecipanti collaborano per sviluppare soluzioni innovative in un tempo limitato, solitamente 48 ore. Ogni hackathon è unico e prevede una serie di caratteristiche ben definite che ne regolano l'organizzazione:

- Organizzazione dell'**Hackathon**:
  L'**Hackathon** viene creato dalla figura dell'**Organizzatore**, un **Utente** registrato sulla piattaforma, che ne definisce i dettagli fondamentali: un *titolo* identificativo, la *sede* in cui si svolgerà, *la durata* (tipicamente due giorni), e parametri operativi come il *numero massimo di partecipanti* e la *dimensione massima dei team*.

- Selezione dei **Giudici**:
  Una volta configurato l'evento, l'**Organizzatore** seleziona un gruppo di **Giudici** tra gli **Utenti** già presenti nella piattaforma, invitandoli a valutare i progetti presentati dai **Partecipanti**.

- Registrazione e formazione dei **Team**:
  Le iscrizioni all'**Hackathon** vengono aperte dall'**Organizzatore** e rimangono disponibili fino a 48 ore prima dell'inizio dell'evento, momento in cui i **Team** formati diventano definitivi. In questo periodo, gli **Utenti** possono registrarsi singolarmente o formare **Team**. Se un **Utente** non è ancora registrato sulla piattaforma, può creare un account in fase di iscrizione.

- Svolgimento dell'**Hackathon**
  All'inizio della competizione, i **Giudici** pubblicano la *descrizione del problema* da risolvere, che servirà come base per lo sviluppo dei progetti. Durante l'evento, i **Team** lavorano in modo indipendente ma sono tenuti a caricare periodicamente aggiornamenti sui loro progressi sotto forma di **Documenti** successivamente valutati dai **Giudici**.

- Valutazione e classifica finale
  Al termine dell'**Hackathon**, ogni **Giudice** assegna un *voto* a ciascun **Team** in base alla qualità della soluzione proposta. La piattaforma raccoglie automaticamente tutti i voti e pubblica una classifica finale resa disponibile a tutti gli **Utenti**.

1.3.2 Vincoli Individuati

Per garantire il corretto funzionamento della sistema, sono state imposte le seguenti regole:

- Un **Utente** può assumere più di un ruolo, ma **non** in una *singola* competizione
- Un **Utente** deve essere registrato all'**Hackathon** prima di poter prendere parte ad un **Team**
- Un **Utente** può unirsi a più **Team**, ma **non** in in *singolo* **Hackathon**
- Un **Team** può essere formato anche da un solo **Partecipante**
- Un gruppo di **Giudici**, per essere considerato valido, deve essere composto da almeno 2 membri
- Un **Hackathon**, per essere considerato valido, deve prevedere almeno 2 **Team**
- Il *Voto* finale deve essere compreso tra 0 e 10
- Prima di poter dare un *Voto* ad un **Team**, un **Giudice** deve *esaminare* ogni documento prodotto da quel team.
- Un **Partecipante** deve rispettare i vincoli temporali impostegli dall'**Organizzatore** per poter effettuare le operazioni che desidera.
- Per essere valutati correttamente, i **Team** devono produrre almeno 1 **Documento**

Formalizzazione dei Vincoli

$\mathcal{G} = \{g_1, \ldots, g_k\}$	$\lvert\mathcal{G}\rvert \geq 2$	Insieme dei Giudici
$\mathcal{P} = \{p_1, \ldots, p_m\}$		Insieme dei Partecipanti
$\mathcal{O} = \{o_1, \ldots, o_n\}$	$\lvert\mathcal{O}\rvert = 1$	Insieme degli Organizzatori, 1 per Hackathon
$\mathcal{T} = \{\, t_0, t_1, \ldots, t_z \mid \lvert t_i\rvert \geq 2,\ i \in \{0, \ldots, z\}\,\}$		Insieme dei Team
$t_i = \{p_{i,1}, \ldots, p_{i,\lvert t_i\rvert}\} \subseteq \mathcal{P}, \quad t_i \cap t_j = \emptyset\ \ \forall i \neq j$		Partecipanti di ogni team, disgiunti tra loro

$$voto \in [0, 10]$$

$\mathcal{H} = \langle \mathcal{P}, \mathcal{O}, \mathcal{G}, \mathcal{T} \rangle$ Istanza di Hackathon

Convenzioni Notazionali

$p_{i,j}$	Partecipante numero j del team t_i
i	Indice del team, $i \in \{0, \ldots, z\}$
j	Posizione del partecipante all'interno del team t_i, $j \in \{1, \ldots, \lvert t_i\rvert\}$

1.3.3 Requisiti Funzionali

Il sistema per la gestione degli hackathon deve soddisfare una serie di funzionalità volte a supportare il ciclo di vita completo degli eventi di programmazione competitiva, dalla creazione dell'evento fino alla pubblicazione delle classifiche finali.

Gestione degli utenti

Registrazione di nuovi utenti sulla piattaforma, specificando:

- Credenziali di accesso
- Dati anagrafici

Autenticazione e gestione del profilo utente.

Gestione degli hackathon

Creazione di un nuovo hackathon da parte di un organizzatore, specificando:

- Titolo identificativo
- Sede dell'evento
- Intervallo temporale (tipicamente 2 giorni)
- Numero massimo di iscritti
- Dimensione massima del team

Visualizzazione e ricerca di hackathon per:

- Titolo
- Sede
- Data
- Organizzatore

Gestione dei giudici

Selezione e invito di giudici tra gli utenti registrati

Pubblicazione della descrizione del problema da affrontare all'inizio dell'hackathon.

Consultazione e commento dei progressi caricati dai team.

Gestione delle registrazioni

Apertura delle registrazioni per un hackathon specifico.

Registrazione dei partecipanti durante il periodo di iscrizione.

Controllo del rispetto del numero massimo di iscritti.

Gestione dei team

Formazione dei team da parte dei partecipanti registrati:

- Creazione di nuovi team
- Adesione ad altri team
- Controllo del rispetto della dimensione massima del team

Finalizzazione automatica dei team alla chiusura delle iscrizioni.

Visualizzazione della composizione dei team.

Gestione dei progressi

Caricamento periodico degli aggiornamenti sui progressi:

- Upload di documenti contenenti lo stato di avanzamento

Consultazione dei progressi da parte dei giudici.

Aggiunta di commenti e feedback da parte dei giudici.

Gestione delle valutazioni

Assegnazione dei voti finali:

- Voto da 0 a 10 per ciascun team da parte di ogni giudice
- Controllo della completezza delle valutazioni
- Calcolo automatico delle medie e classifiche

Pubblicazione delle classifiche finali una volta acquisiti tutti i voti.

Consultazione dei risultati e delle statistiche dell'evento.

2 Progettazione Concettuale

2.1 Entità e Relazioni

Gli oggetti del dominio sono stati modellati attraverso l'individuazione di **Entità** e **Relazioni**, con i rispettivi attributi.

2.1.1 Entità

Utente

Rappresenta tutti gli attori del sistema:

- Nome
- Cognome
- Username
- Password

Partecipante

Giudice

Organizzatore

Team

Gruppi di lavoro dei partecipanti:

- Nome
- NumeroMembri

Documento

I documenti prodotti dai team:

- Titolo
- Contenuto
- Commento

Hackathon

Evento principale del sistema:

- Titolo
- Sede
- Durata
- DataInizio
- DataFine
- PeriodoIscrizioni
- DataAperturaIscrizioni
- DataChiusuraIscrizioni
- MaxIscritti
- MaxDimTeam
- DescrizioneProblema

2.1.2 Relazioni

Le relazioni individuate che intercorrono tra le entità del sistema:

Registrazione

- Data

Partecipazione

Competizione

Organizzazione

Pubblicazione

Votazione

- Voto

Selezione

Esaminazione

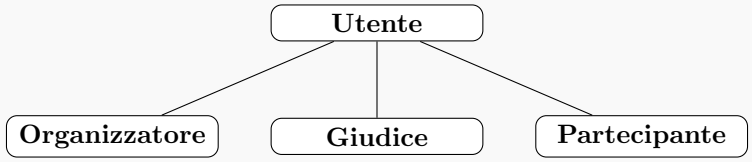
Convenzioni Notazionali

Nome → Entità o Relazione
Nome → Attributo dell'Entità o della Relazione
Attributo → Attributo Identificativo dell'Entità
Attributo → Attributo Identificativo **Parziale** dell'Entità

2.2 Gerarchie tra Entità

Durante la progettazione concettuale è stata individuata l'entità generale **Utente**, dalla quale derivano tre entità in specializzazione: **Organizzatore**, **Giudice** e **Partecipante**. Tale gerarchia consente di modellare in maniera più accurata i diversi ruoli che gli utenti possono assumere all'interno di un **Hackathon**.

Diagramma della Gerarchia



2.2.1 Vincoli della Gerarchia

I vincoli che caratterizzano la gerarchia sono:
- **Vincolo di Overlapping**: un'istanza può appartenere a *più* di una sottoclasse.
- **Vincolo di Completezza Parziale**: un'istanza della superclasse non è obbligata a far parte di una sottoclasse.

Questo vincolo riguarda la base di dati nella sua interezza ma, come specificato precedentemente, nel contesto di un singolo **Hackathon** assume la forma di un **vincolo di disgiunzione**.
Tali vincoli garantiscono il corretto funzionamento del sistema, in accordo con l'analisi dei requisiti:

- Un **Utente** "diventa" **Partecipante** se e solo se ($\Leftrightarrow$) completa la registrazione ad un **Hackathon**. Gli utenti generici non assumono automaticamente questo status.
- I **Giudici** hanno un ruolo esclusivo: possono valutare i progetti ma non iscriversi come partecipanti.
- Gli **Organizzatori** hanno accesso solo alle funzioni di gestione e non possono partecipare direttamente alla competizione.

2.3 Scelte di Progettazione Concettuale

Per rendere il sistema quanto più aderente al contesto reale, le seguenti entità e relative relazioni sono state progettate in modo da consentire la gestione dello **storico degli eventi e delle interazioni**:

- **Partecipante**: tramite le relazioni **Registrazione** e **Partecipazione** memorizziamo lo storico di partecipazioni agli hackathon passati ed attuali, senza dover vincolare gli utenti a giocare in una sola competizione alla volta.

- **Organizzatore**: con la relazione di **Organizzazione** conserviamo i record di gestione delle varie gare, evitando di avere un **Utente** vincolato ad un **Hackathon** singolo.

- **Giudice**: grazie alle relazioni **Votazione** ed **Esaminazione** conserviamo trasversalmente nei vari **Hackathon** le valutazioni emesse dai giudici, grazie al fatto che un **Team** è univocamente legato al proprio **Hackathon** di riferimento.

- **Selezione**: questa relazione ternaria consente di conservare le varie istanze di elezione di **Giudici** dagli **Organizzatori** anche tramite stessi **Utenti**, grazie al collegamento con **Hackathon** che consente di discernere univocamente un record dall'altro, conservando così lo storico di inviti.

Questo approccio consente di:
- mantenere una **tracciabilità temporale** delle attività
- permettere **analisi retrospettive** e la generazione di report
- garantire **scalabilità** e supporto a future evoluzioni della piattaforma

Dal punto di vista pratico sarà quindi possibile ottenere:
- per ogni **Partecipante**: a quali **Team** ha preso parte e in quali **Hackathon**
- per ogni **Organizzatore**: quali **Hackathon** ha gestito e quali **Giudici** ha invitato
- per ogni **Giudice**: l'elenco di valutazioni che ha emesso in ogni **Hackathon**

In questo modo la progettazione garantisce **coerenza**, **completezza** e **scalabilità**. È doveroso aggiungere che su richiesta di un utente o a seguito di *manutenzione periodica* della base di dati, questi record aggiuntivi preservati per lo storico possano essere **eliminati** per garantire il funzionamento del sistema nel tempo.

Inoltre, durante questa fase di progettazione concettuale si è scelto di includere degli attributi **non esplicitamente richiesti** dalla traccia, al fine di meglio esprimere i concetti e le necessità che sono intrinseche in una richiesta come quella del problema in esame, in particolare:

- Nome — Cognome — Username — Password
  Questi attributi sono stati inseriti nella fase concettuale per esprimere il concetto di "*utenti iscritti alla piattaforma*", per memorizzarne le credenziali di accesso e simulare un sistema di account.

- Nome — NumeroMembri
  Questi attributi sono presenti per identificare in modo più esplicativo un **Team**, presentando informazioni semplici e con poco carico sul sistema; un buon compromesso tra richiesta del cliente e chiarezza della base di dati.

- Data
  Un attributo utile a dare rilevanza ai record riferiti ad una **Registrazione** da parte di un **Utente**. Memorizza la data precisa nella finestra di tempo utile per registrarsi nel quale l'**Utente** ha effettuato tale operazione.

- DataInizio — DataFine — DataAperturaIscrizioni — DataChiusuraIscrizioni
  I suddetti attributi esprimono più chiaramente le specifiche date necessarie a tracciare la *durata* della competizione e il suo relativo *periodo utile per le registrazioni.*
  Inoltre, l'attributo *DataChiusuraIscrizioni* potrebbe essere considerato come *derivabile*, poiché la richiesta del cliente sottolinea che le iscrizioni si chiudano `"2 giorni prima dell'evento"`. Potremmo quindi effettuare una semplice sottrazione ed ottenere la *DataInizio*, ma si è scelto di rilassare questo vincolo per permettere al sistema una maggiore flessibilità, a discrezione dell'**Organizzatore**.

Relazione Ternaria (M:N:P)

La relazione di **Selezione** è espressa tramite una rara ma utile opzione a disposizione dei progettisti di basi di dati: una **relazione n-aria**. Questa associazione è frutto di un'estensione di quella binaria tra **Organizzatore** e **Giudice**, che non presenterebbe modo di rintracciare l'**Hackathon** di riferimento per via dell'implementazione dello *storico degli eventi.* Così facendo introduciamo un legame univoco per ogni diverso **Hackathon**, permettendo che lo stesso **Organizzatore** inviti gli stessi **Giudici** in diverse competizioni.
È stato scelto, inoltre, di **NON reificare** l'associazione in quanto un'entità *Selezione* sarebbe stata troppo lontana da quello che si evince dal mini-world, descrivendo un aspetto poco riconducibile ad una vera e propria entità del sistema.

2.4 Dizionario delle Entità

Entità	Attributi	Descrizione
Utente	- `Nome` - `Cognome` - `Username` - `Password`	Persona iscritta alla piattaforma che può prendere parte agli eventi disponibili
Partecipante	–	Membro di un **Team** che partecipa all'**Hackathon**
Giudice	–	**Utente** selezionato che può giudicare e valutare gli elaborati delle squadre
Organizzatore	–	Definisce le informazioni necessarie per pubblicare un **Hackathon**
Team	- `Nome` - `NumeroMembri`	Squadra di **Partecipanti** che ha il compito di pubblicare i progressi durante la competizione
Documento	- `Titolo` - `Contenuto` - `Commento`	File prodotto dai vari **Team** che verrà poi valutato dai **Giudici**
Hackathon	- `Titolo` - `Sede` - `Durata` - `DataInizio` - `DataFine` - `PeriodoIscrizioni` - `DataAperturaIscrizioni` - `DataChiusuraIscrizioni` - `MaxIscritti` - `MaxDimTeam` - `DescrizioneProblema`	Competizione istituita da un **Organizzatore**

Tabella 1: Descrizione delle entità del sistema

2.5 Dizionario delle Relazioni

Relazioni	Tipologia	Descrizione
Competizione	Uno-a-Molti	- Un Team compete in un 1 singolo Hackathon - In 1 Hackathon competono nessuno o più Team
Pubblicazione	Uno-a-Molti	- Un Team pubblica nessuno o più Documenti - Un Documento è pubblicato da 1 ed un solo Team
Esaminazione	Molti-a-Molti	- Un Giudice esamina nessuno o più Documenti - Un Documento è esaminato da nessuno o più Giudici
Organizzazione	Uno-a-Molti	- Un Organizzatore organizza 1 o più Hackathon - Un Hackathon è organizzato da 1 Organizzatore
Selezione	Molti-a-Molti-a-Molti	- Un Organizzatore seleziona 2 o più Giudici nel contesto di 1 o più Hackathon - Un Giudice è selezionato da 1 o più Organizzatori nel contesto di 1 o più Hackathon - In un Hackathon sono selezionati 2 o più Giudici da 1 Organizzatore
Partecipazione	Molti-a-Molti	- Un Partecipante partecipa ad 1 o più Team - Un Team possiede 1 o più Partecipanti
Registrazione	Molti-a-Molti	- **Data**: attributo di relazione per memorizzare la data di Registrazione all'Hackathon - Un Partecipante si registra a 1 o più Hackathon - Ad un Hackathon si registrano nessuno o più Partecipanti
Votazione	Molti-a-Molti	- **Voto**: attributo di relazione che individua il valore numerico della votazione - Un Team è valutato da nessuno o più Giudici - Un Giudice valuta nessuno o più Team

Tabella 2: Descrizione delle relazioni tra le entità individuate

2.6 Diagramma Concettuale — ER

Definizione: Tale *Diagramma Entity–Relationship (ER)* è un **modello concettuale** utilizzato per rappresentare in forma grafica le entità individuate di dominio, i loro attributi e le relazioni che intercorrono tra di esse.
Questo nella presente fase di progettazione concettuale, consente di descrivere la struttura dei dati in modo indipendente dall'implementazione fisica che avverrà successivamente.

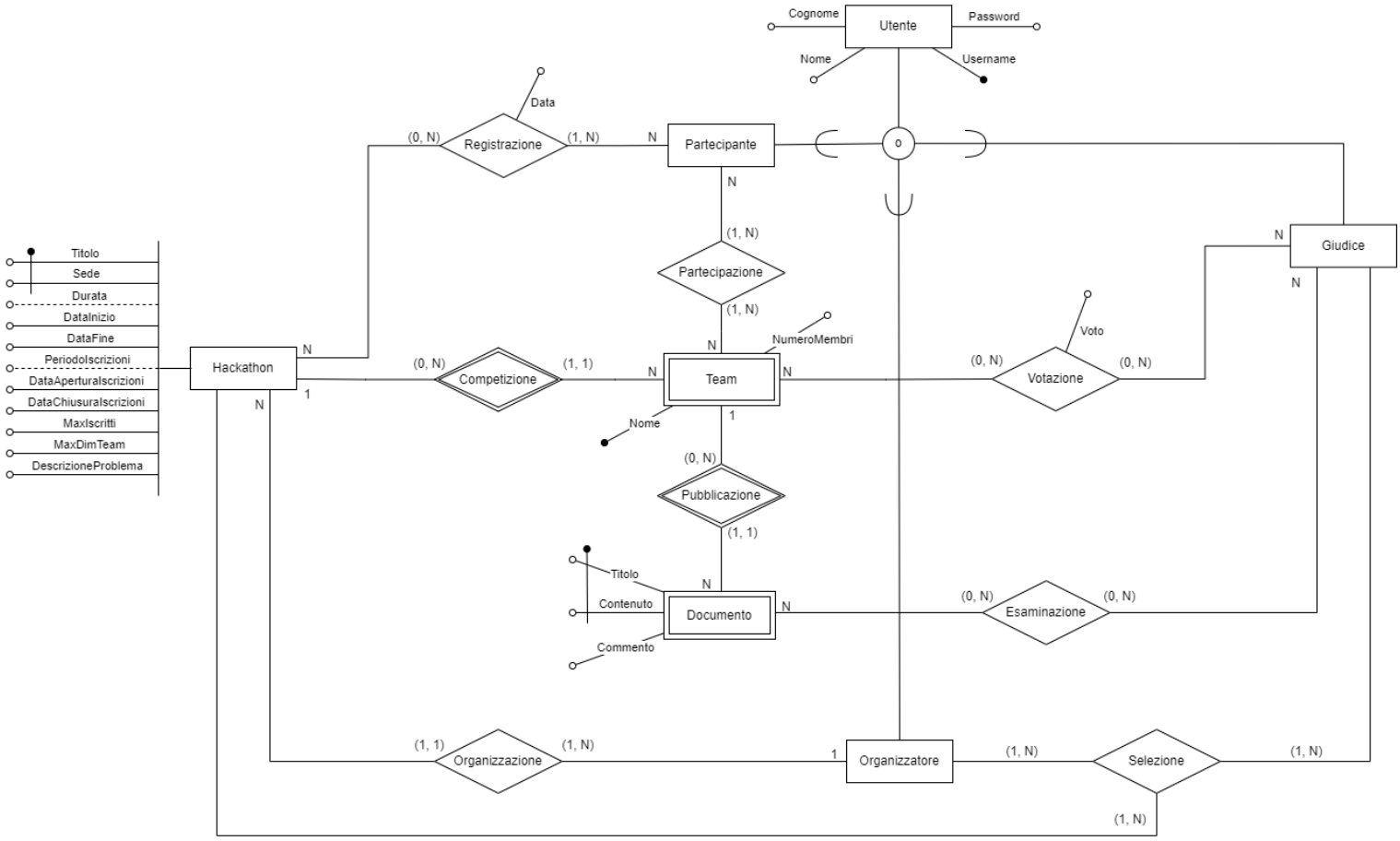
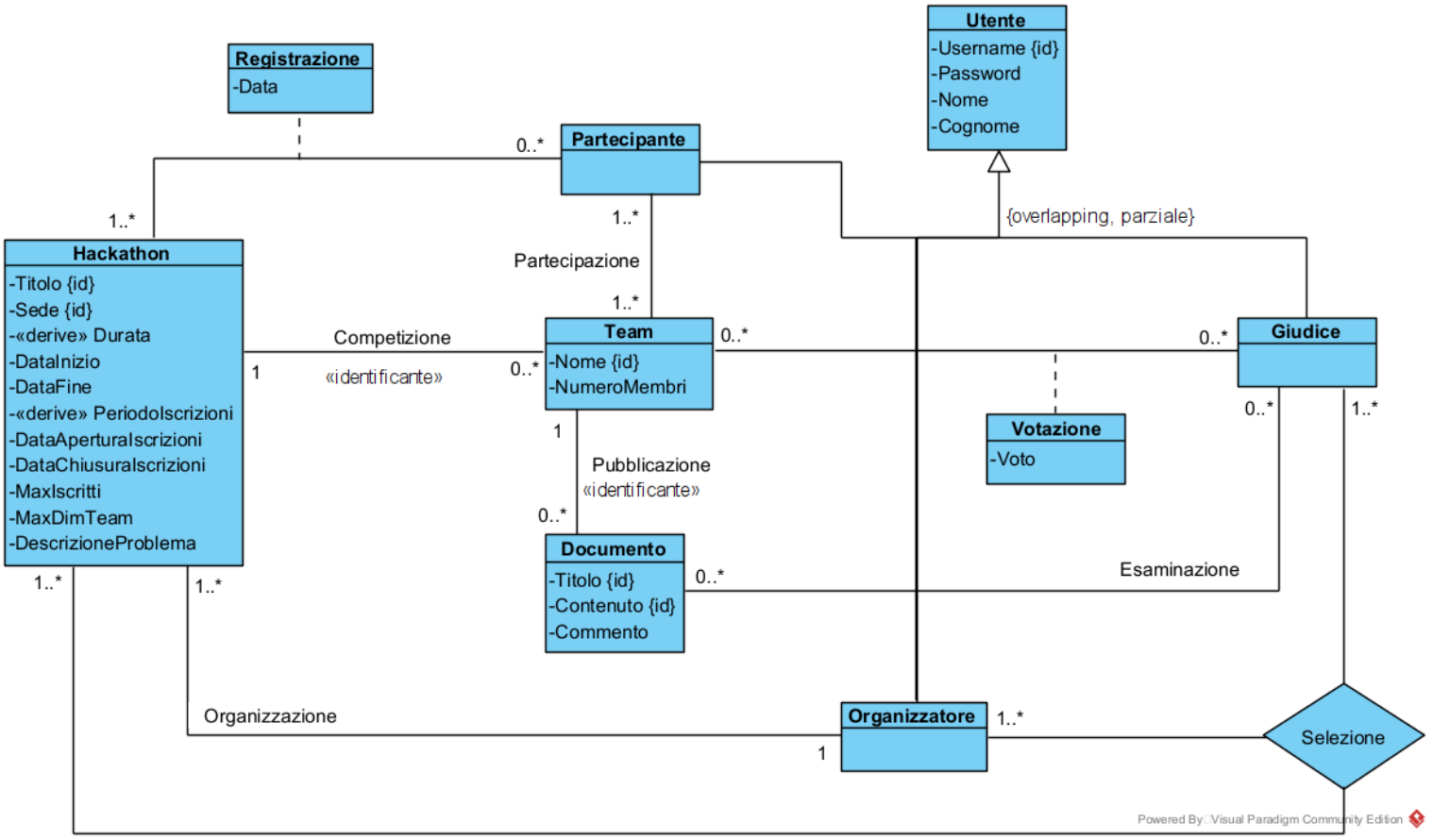


Figura 1: Stereotipizzazione Atzeni, Ceri, Paraboschi, Fraternali, Torlone

2.7 Diagrama Concettuale — UML

Tale versione del **modello concettuale** è espresso nei termini e formalismi descritti dall'**Unified Modeling Language**.



Convenzioni Notazionali

Attributo `{id}` → Attributo Identificatore

2.8 Ristrutturazione del Modello Concettuale

Introduzione

> *Definizione*: La determinazione del modello concettuale precedentemente illustrata deve essere seguita da un processo di traduzione tale da portare a stabilire uno **schema logico**. Quest'ultimo processo è definito *ristrutturazione del modello concettuale*, una fase il cui obiettivo è quello di definire uno schema logico in grado di descrivere tutte le informazioni contenute nello schema concettuale prodotto dalla fase di progettazione concettuale.

Un modello logico infatti, presenta delle *restrizioni* rispetto al modello concettuale in quanto agli elementi rappresentabili in esso.
Degli elementi costituenti un modello concettuale i seguenti non sono rappresentabili nel logico:

- **Gerarchie**
- **Attributi Multipli**
- **Attributi Strutturati**

Tale fase di ristrutturazione si basa su:

- Criteri di ottimizzazione dello schema
- Regole di traduzione verso il modello logico

A partire dal **diagramma concettuale** e dal **carico applicativo** eventuale, si ottiene attraverso i seguenti step, il **diagramma ristrutturato**.

Fasi del Processo di Ristrutturazione

Il processo di ristrutturazione si articola nelle seguenti fasi operative:

1. Analisi delle Ridondanze
2. Eliminazione delle Generalizzazioni
3. Eliminazione Attributi Multivalore
4. Eliminazione Attributi Strutturati
5. Partizionamento/Accorpamento di Entità e Associazioni
6. Scelta degli Identificatori Principali

2.8.1 Analisi delle Ridondanze

> *Definizione*: Una ridondanza in uno schema concettuale corrisponde alla presenza di un dato duplicato e/o derivabile da altri dati.

Tale fase di analisi si prefigge l'obiettivo di individuare informazioni ritenibili tali e successivamente applicare una delle seguenti tecniche di risoluzione:

- **Mantenimento della ridondanza**: Le ridondanze individuate vengono mantenute per motivi di efficienza (es. campi frequentemente consultati).
- **Eliminazione della ridondanza**: Le ridondanze individuate vengono rimosse poiché potrebbero causare errori e/o aumentare i costi di aggiornamento.

Dato derivabile

> *Definizione*: Un dato derivabile è un'informazione che può essere ottenuta attraverso operazioni di calcolo, elaborazione o interrogazione su altri dati **già presenti nella base di dati**. La sua memorizzazione esplicita rappresenta una forma di ridondanza.

Individuazione Dati Derivabili:

Nel contesto di una progettazione concettuale, è possibile che

1. Attributi e
2. Relazioni

vengano ritenuti possibilmente derivabili.

Attributo Derivabile

> *Definizione*: Un attributo derivabile è un attributo il cui valore può essere calcolato o determinato a partire dai valori di altri attributi della stessa entità o di entità correlate attraverso operazioni quali:
>
> - Operazioni aritmetiche (somma, differenza, prodotto, divisione)
> - Funzioni di aggregazione (successivamente illustrate)
> - Operazioni temporali (differenze tra date, calcoli di età)

Individuazione Attributi Derivabili

Gli attributi di dominio ritenuti **derivabili** individuati sono i seguenti:

1. **NumeroMembri**
2. **PeriodoIscrizioni**
3. **Durata**

NumeroMembri Durante la progettazione concettuale è stata presa in considerazione la possibilità di includere nell'elenco degli attributi dell'entità *Team*, l'attributo `NumeroMembri`, che sta ad indicare il numero di membri componenti un determinato Team.
Quest'ultimo, nella fase di ristrutturazione è stato classificato come *derivabile* in quanto sarebbe necessario effettuare il solo conteggio dei membri di un Team. Potrebbe essere una potenziale **ridondanza**.

PeriodoIscrizioni In secondo luogo, `PeriodoIscrizioni` è un attributo specificato dal dominio e rappresenta l'intervallo di tempo che va dall'apertura delle iscrizioni dell'*hackathon* fino alla loro chiusura.
È stato ritenuto quindi anch'esso *derivabile*, in quanto calcolabile tramite la differenza dei due attributi sovraelencati. Anch'esso, quindi, potrebbe essere una potenziale **ridondanza**.

Durata Anche `Durata` è un attributo specificato dalla richiesta del cliente e descrive la durata in giorni dell'intera competizione.
Come nel caso precedente quindi, sarebbe possibile *derivare* questo attributo dalla differenza delle due date di inizio e fine gara. È quindi di fatto una potenziale **ridondanza**.

Per determinare se questi attributi costituiscano:

- una **ridondanza effettiva** *oppure*
- rappresentino **un'informazione rilevante da mantenere**

è stata costruita la seguente **tavola d'analisi**, tale per cui sarà possibile scegliere uno dei metodi di risoluzione per le ridondanze da applicare.

Tavole d'Analisi - Attributi Derivabili

Costo degli Accessi

Lettura (L): 1 unità di costo
Scrittura (S): 2 unità di costo

1. **NumeroMembri**

Sono state prese in considerazione due azioni ritenute potenzialmente ricorrenti nell'utilizzo della piattaforma:

- Partecipazione di un membro a un Team (consultazione del numero di membri)
- Visualizzazione dei dati relativi a un Team

Valutazione degli Indici di Prestazione

- **Costo di Memorizzazione del Dato Ridondante**
  - Numero di Team nel database: 100
  - Dimensione dell'attributo `NumeroMembri`: 4 byte (intero)

$$\text{Mem(NumeroMembri)} = 4 \text{ byte} \times 100 \text{ team} = 400 \text{ byte} \tag{1}$$

Tavole degli Accessi

CON RIDONDANZA

Operazione 1: Aggiunta di un nuovo membro a un Team

Entità/Relazione	Tipo	Accessi
Team	E	1 L (verifica esistenza team)
Partecipa	R	1 S (memorizzazione nuova partecipazione)
Team	E	1 S (incremento `NumeroMembri`)

Frequenza: 50 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op1} = (1L + 2S) \times 50 = (1 + 2 \times 2) \times 50 = 250 \text{ accessi/giorno} \tag{2}$$

Operazione 2: Visualizzazione dati di un Team (incluso numero membri)

Entità/Relazione	Tipo	Accessi
Team	E	1 L (lettura diretta di `NumeroMembri`)

Frequenza: 200 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op2} = 1L \times 200 = 200 \text{ accessi/giorno} \tag{3}$$

Costo totale con ridondanza:

$$\text{Costo}_{\text{CON}} = 250 + 200 = 450 \text{ accessi/giorno} \tag{4}$$

SENZA RIDONDANZA

Operazione 1: Aggiunta di un nuovo membro a un Team

Entità/Relazione	Tipo	Accessi
Team	E	1 L (verifica esistenza team)
Partecipa	R	1 S (memorizzazione nuova partecipazione)

(Non si accede a Team per aggiornare il dato derivato)
Costo giornaliero:

$$\text{Costo}_{Op1} = (1L + 1S) \times 50 = (1 + 2) \times 50 = 150 \text{ accessi/giorno} \tag{5}$$

Operazione 2: Visualizzazione dati di un Team (incluso numero membri)

Entità/Relazione	Tipo	Accessi
Team	E	1 L (lettura dati team base)
Partecipa	R	15 L (calcolo COUNT dei membri)

Assumendo una media di 15 membri per team.
Costo giornaliero:

$$\text{Costo}_{Op2} = (1L + 15L) \times 200 = 16 \times 200 = 3200 \text{ accessi/giorno} \tag{6}$$

Costo totale senza ridondanza:

$$\text{Costo}_{\text{SENZA}} = 150 + 3200 = 3350 \text{ accessi/giorno} \tag{7}$$

Confronto e Conclusioni

Analisi Quantitativa

$$\begin{aligned} \text{Risparmio accessi} &= \text{Costo}_{\text{SENZA}} - \text{Costo}_{\text{CON}} && (8)\\ &= 3350 - 450 = 2900 \text{ accessi/giorno} && (9) \end{aligned}$$

$$\text{Costo aggiuntivo memoria} = 400 \text{ byte} \approx 0.4 \text{ KB} \tag{10}$$

Rapporto Benefici/Costi

$$\frac{\text{Risparmio accessi}}{\text{Costo memoria}} = \frac{2900 \text{ accessi/giorno}}{0.4 \text{ KB}} = 7250 \frac{\text{accessi/giorno}}{\text{KB}} \tag{11}$$

In conclusione, la memorizzazione dell'attributo **NumeroMembri** risulta **vantaggiosa** poiché:

1. **Riduzione drastica degli accessi**: da 3350 a 450 accessi/giorno (-86.6%)
2. **Costo di memoria trascurabile**: solo 400 byte aggiuntivi
3. **Miglioramento prestazioni**: le operazioni di lettura (molto frequenti) sono significativamente più veloci
4. **Rapporto costo-beneficio ottimale**: il risparmio in accessi compensa ampiamente il costo in memoria

L'attributo **NumeroMembri** rappresenta quindi una **ridondanza utile** e **non dannosa** poiché:

$$\text{Benefici}_{\text{prestazioni}} \gg \text{Costi}_{\text{memoria}} \tag{12}$$

2. PeriodoIscrizioni

È stata presa in considerazione la seguente azione ritenuta potenzialmente ricorrente nell'utilizzo della piattaforma:

- Visualizzazione delle informazioni relative a un Hackathon

Valutazione degli Indici di Prestazione

- **Costo di Memorizzazione del Dato Ridondante**
  - Numero di Hackathon nel database: 50
  - Dimensione dell'attributo `PeriodoIscrizioni`: 4 byte (intero, giorni)

$$\text{Mem(PeriodoIscrizioni)} = 4 \text{ byte} \times 50 \text{ hackathon} = 200 \text{ byte} \tag{13}$$

Tavole degli Accessi

CON RIDONDANZA

Operazione 1: Visualizzazione informazioni di un Hackathon (incluso periodo iscrizioni)

Entità/Relazione	Tipo	Accessi
Hackathon	E	1 L (lettura diretta di `PeriodoIscrizioni`)

Frequenza: 150 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op2} = 1L \times 150 = 150 \text{ accessi/giorno} \tag{14}$$

Costo totale con ridondanza:

$$\text{Costo}_{\text{CON}} = 150 \text{ accessi/giorno} \tag{15}$$

SENZA RIDONDANZA

Operazione 1: Visualizzazione informazioni di un Hackathon (incluso periodo iscrizioni)

Entità/Relazione	Tipo	Accessi
Hackathon	E	2 L (lettura `DataAperturaIscrizioni` e `DataChiusuraIscrizioni`)

(Il calcolo del periodo viene effettuato in memoria tramite differenza tra le date)
Costo giornaliero:

$$\text{Costo}_{Op2} = 2L \times 150 = 300 \text{ accessi/giorno} \tag{16}$$

Costo totale senza ridondanza:

$$\text{Costo}_{\text{SENZA}} = 300 \text{ accessi/giorno} \tag{17}$$

Confronto e Conclusioni

Analisi Quantitativa

$$\begin{aligned} \text{Differenza accessi} &= \text{Costo}_{\text{SENZA}} - \text{Costo}_{\text{CON}} && (18)\\ &= 300 - 150 = 150 \text{ accessi/giorno} && (19) \end{aligned}$$

$$\text{Costo aggiuntivo memoria} = 200 \text{ byte} \approx 0.2 \text{ KB} \tag{20}$$

Rapporto Benefici/Costi

$$\frac{\text{Risparmio accessi}}{\text{Costo memoria}} = \frac{150 \text{ accessi/giorno}}{0.2 \text{ KB}} = 750 \frac{\text{accessi/giorno}}{\text{KB}} \tag{21}$$

In conclusione, la memorizzazione dell'attributo **PeriodoIscrizioni** risulta **svantaggiosa** per le seguenti motivazioni:

1. **Assenza di operazioni rilevanti**: l'attributo manca di sostanziali *use cases* per poter giustificarne la memorizzazione
2. **Ridotto risparmio di accessi**: il vantaggio guadagnato è direttamente proporzionale alla frequenza dell'operazione di visualizzazione dei dati di un hackathon
3. **Ridondanza inutile**: il calcolo della differenza tra date è computazionalmente trascurabile
4. **Mantenimento della consistenza**: eliminando la ridondanza si evitano possibili inconsistenze tra i dati

L'attributo **PeriodoIscrizioni** rappresenta quindi una **ridondanza dannosa** e **non necessaria** poiché:

$$\text{Benefici}_{\text{prestazioni}} > \text{Costi}_{\text{memoria}} + \text{Costi}_{\text{consistenza}} \not\rightarrow \text{Rilevanza}_{\text{informazione}} \tag{22}$$

Ovverosia, l'attributo sarebbe formalmente memorizzabile ma non ne conseguono vantaggi rilevanti in termini di operazioni frequenti, dato il poco utilizzo del dato.

3. Durata Sono state prese in considerazione due azioni ritenute potenzialmente ricorrenti nell'utilizzo della piattaforma:

- Creazione di un nuovo Hackathon
- Ricerca Hackathon per durata specifica
- Consultazione statistiche per durata

Costo di Memorizzazione del Dato Ridondante

$$\text{Mem(Durata)} = 4 \text{ byte} \times 50 \text{ hackathon} = 200 \text{ byte} \tag{23}$$

Tavole degli Accessi

CON RIDONDANZA

Operazione 1: Creazione di un nuovo hackathon

Entità/Relazione	Tipo	Accessi
Hackathon	E	1 S (inserimento con durata precalcolata)

Frequenza: 5 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op1} = 1S \times 5 = 2 \times 5 = 10 \text{ accessi/giorno} \tag{24}$$

—

Operazione 2: Ricerca hackathon per durata specifica

Entità/Relazione	Tipo	Accessi
Hackathon	E	10 L (ricerca diretta su indice Durata)

Assumendo 10 occorrenze di hackathon con la durata ricercata.
Frequenza: 20 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op2} = 10L \times 20 = 10 \times 20 = 200 \text{ accessi/giorno} \tag{25}$$

—

Operazione 3: Consultazione statistiche per durata

Entità/Relazione	Tipo	Accessi
Hackathon	E	50 L (scansione completa degli hackathon)

Frequenza: 5 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op3} = 50L \times 5 = 50 \times 5 = 250 \text{ accessi/giorno} \tag{26}$$

Costo totale con ridondanza:

$$\text{Costo}_{CON} = 10 + 200 + 250 = 460 \text{ accessi/giorno} \tag{27}$$

—

SENZA RIDONDANZA

Operazione 1: Creazione di un nuovo hackathon

Entità/Relazione	Tipo	Accessi
Hackathon	E	0 S (inserimento senza calcolo durata)

Frequenza: 5 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op1} = 0S \times 5 = 0 \times 5 = 0 \text{ accessi/giorno} \tag{28}$$

—

Operazione 2: Ricerca hackathon per durata specifica

Entità/Relazione	Tipo	Accessi
Hackathon	E	50×2 L (scansione completa per `DataInizio` e `DataFine`)

Frequenza: 20 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op2} = 100L \times 20 = 100 \times 20 = 2000 \text{ accessi/giorno} \tag{29}$$

—

Operazione 3: Consultazione statistiche eventi per durata

Entità/Relazione	Tipo	Accessi
Hackathon	E	50×2 L (scansione completa per `DataInizio` e `DataFine`)

Frequenza: 5 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op3} = 100L \times 5 = 100 \times 5 = 500 \text{ accessi/giorno} \tag{30}$$

Costo totale senza ridondanza:

$$\text{Costo}_{SENZA} = 0 + 2000 + 500 = 2500 \text{ accessi/giorno} \tag{31}$$

—

Confronto e Conclusioni

Analisi Quantitativa

$$\text{Risparmio accessi} = \text{Costo}_{SENZA} - \text{Costo}_{CON} \tag{32}$$

$$= 2500 - 460 = 2040 \text{ accessi/giorno} \tag{33}$$

$$\text{Costo aggiuntivo memoria} = 200 \text{ byte} \approx 0.2 \text{ KB} \tag{34}$$

Rapporto Benefici/Costi

$$\frac{\text{Risparmio accessi}}{\text{Costo memoria}} = \frac{2040 \text{ accessi/giorno}}{0.2 \text{ KB}} = 10200 \ \frac{\text{accessi/giorno}}{\text{KB}} \tag{35}$$

—

Conclusione La memorizzazione dell'attributo **Durata** continua a risultare **vantaggiosa**:

- Riduzione significativa degli accessi: da 2500 a 460 accessi/giorno (-81,6%)
- Costo di memoria trascurabile: solo 200 byte
- Ottimizzazione delle ricerche e delle aggregazioni statistiche

Relazione Derivabile

Definizione: Una relazione derivabile è una relazione che può essere ricostruita attraverso la composizione di altre relazioni esistenti nel database mediante operazioni di:

- Join multipli attraverso percorsi di relazioni intermedie
- Operazioni insiemistiche ($\cup$, $\cap$, $\setminus$)
- Proiezioni e selezioni su relazioni esistenti (σ, π)
- Aggregazioni su strutture relazionali complesse

Individuazione Relazioni Derivabili

La relazione presente nella progettazione concettuale ritenuta derivabile è la seguente:

- **Registrazione**

$$\textit{Hackathon} \xrightarrow{\text{Registrazione}} \textit{Partecipante}$$

La relazione in questione presenta una potenziale criticità che richiede un'attenta valutazione progettuale. Dal punto di vista della teoria relazionale, questa relazione potrebbe essere considerata ridondante poiché derivabile attraverso la composizione delle relazioni già esistenti nel modello dati.
Formalmente, la relazione **Registrazione** potrebbe essere espressa ugualmente tramite Il seguente percorso di derivazione alternativo:

$$\textit{Hackathon} \xrightarrow{\text{Competizione}} \textit{Team} \xrightarrow{\text{Partecipazione}} \textit{Partecipante}$$

Tutto ciò solleva interrogativi fondamentali sulla necessità di mantenere esplicitamente la relazione **Registrazione**. L'eliminazione di tale relazione consentirebbe di:

- Ridurre lo spazio di memorizzazione occupato
- Eliminare potenziali inconsistenze dovute alla ridondanza
- Semplificare la struttura del modello dati

Tuttavia, questa valutazione puramente teorica potrebbe non considerare adeguatamente gli impatti operativi che deriverebbero dall'adozione di un approccio basato esclusivamente sulla derivazione delle informazioni di registrazione.
La questione centrale da risolvere riguarda quindi determinare se la relazione **Registrazione** costituisca effettivamente una ridondanza dannosa da eliminare oppure se rappresenti una scelta progettuale giustificata.
La seguente tavola d'analisi si propone di fornire una valutazione quantitativa e qualitativa per guidare questa decisione progettuale critica.

Tavole d'Analisi - Relazioni Derivabili

Registrazione

- Registrazione di un Partecipante ad un Hackathon (abilitazione alla partecipazione)
- Visualizzazione dei partecipanti registrati ad un Hackathon
- Verifica dell'abilitazione di un partecipante per la formazione di team

Valutazione degli Indici di Prestazione:

- **Costo di Memorizzazione della Relazione Ridondante**
  - Numero di registrazioni nel database: 1500 (30 partecipanti × 50 hackathon)
  - Dimensione record: 12 byte (2 chiavi esterne da 4 byte ciascuna più la data da 4 byte)

$$\text{Mem(Registrazione)} = 12 \text{ byte} \times 1500 \text{ registrazioni} = 18000 \text{ byte} = 18 \text{ KB} \tag{36}$$

- **Costo degli Accessi**
  Un accesso in **scrittura** ha costo doppio rispetto ad un accesso in **lettura**:
  - Accesso in Lettura (L) = 1 unità di costo
  - Accesso in Scrittura (S) = 2 unità di costo

Tavole degli Accessi

CON RELAZIONE RIDONDANTE

Operazione 1: Registrazione di un Partecipante ad un Hackathon

Entità/Relazione	Tipo	Accessi
Hackathon	E	1 L (per verificare esistenza e stato)
Partecipante	E	1 L (per verificare esistenza partecipante)
Registrazione	R	1 S (per memorizzare registrazione diretta)

Frequenza: 200 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op1} = (2L + 1S) \times 200 = (2 + 2) \times 200 = 800 \text{ accessi/giorno} \tag{37}$$

Operazione 2: Visualizzazione partecipanti registrati ad un Hackathon

Entità/Relazione	Tipo	Accessi
Registrazione	R	30 L (lettura diretta registrazioni per hackathon)
Partecipante	E	30 L (dettagli partecipanti registrati)

Frequenza: 50 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op2} = (30L + 30L) \times 50 = 60 \times 50 = 3000 \text{ accessi/giorno} \tag{38}$$

Operazione 3: Verifica abilitazione partecipante per formazione team

Entità/Relazione	Tipo	Accessi
Registrazione	R	1 L (verifica registrazione diretta)

Frequenza: 100 operazioni/giorno
Costo giornaliero:

$$\text{Costo}_{Op3} = 1L \times 100 = 100 \text{ accessi/giorno} \tag{39}$$

Costo totale con relazione ridondante:

$$\text{Costo}_{CON} = 800 + 3000 + 100 = 3900 \text{ accessi/giorno} \tag{40}$$

SENZA RELAZIONE RIDONDANTE

Operazione 1: Registrazione di un Partecipante ad un Hackathon

Entità/Relazione	Tipo	Accessi
Hackathon	E	1 L (per verificare esistenza e stato)
Partecipante	E	1 L (per verificare esistenza partecipante)

(Nessuna memorizzazione diretta della relazione)
Costo giornaliero:

$$\text{Costo}_{Op1} = 2L \times 200 = 2 \times 200 = 400 \text{ accessi/giorno} \tag{41}$$

Operazione 2: Visualizzazione partecipanti registrati ad un Hackathon

Entità/Relazione	Tipo	Accessi
Competizione	R	5 L (team che competono nell'hackathon)
Team	E	5 L (dettagli team)
Partecipazione	R	30 L (6 membri × 5 team)
Partecipante	E	30 L (dettagli partecipanti)

(Percorso: Hackathon → Competizione → Team → Partecipazione → Partecipante)
Costo giornaliero:

$$\text{Costo}_{Op2} = (5L + 5L + 30L + 30L) \times 50 = 70 \times 50 = 3500 \text{ accessi/giorno} \tag{42}$$

Operazione 3: Verifica abilitazione partecipante per formazione team

Entità/Relazione	Tipo	Accessi
Competizione	R	5 L (ricerca team dell'hackathon)
Partecipazione	R	30 L (ricerca partecipante nei team)

(Verifica indiretta attraverso l'appartenenza a team esistenti)
Costo giornaliero:

$$\text{Costo}_{Op3} = (5L + 30L) \times 100 = 35 \times 100 = 3500 \text{ accessi/giorno} \tag{43}$$

Costo totale senza relazione ridondante:

$$\text{Costo}_{SENZA} = 400 + 3500 + 3500 = 7400 \text{ accessi/giorno} \tag{44}$$

Confronto e Conclusioni

Analisi Quantitativa

$$\begin{aligned} \text{Risparmio accessi} &= \text{Costo}_{SENZA} - \text{Costo}_{CON} && (45)\\ &= 7400 - 3900 = 3500 \text{ accessi/giorno} && (46) \end{aligned}$$

$$\text{Costo aggiuntivo memoria} = 12000 \text{ byte} = 12 \text{ KB} \tag{47}$$

Rapporto Benefici/Costi

$$\frac{\text{Risparmio accessi}}{\text{Costo memoria}} = \frac{3500 \text{ accessi/giorno}}{12 \text{ KB}} = 291,6 \frac{\text{accessi/giorno}}{\text{KB}} \tag{48}$$

Considerazioni Architetturali Oltre ai benefici prestazionali quantificabili, la relazione **Registrazione** fornisce vantaggi architetturali significativi:

1. **Livello di astrazione superiore**: La registrazione rappresenta un concetto logico distinto dalla partecipazione in team
2. **Gestione degli stati**: Possibilità di avere partecipanti registrati ma non ancora assegnati a team
3. **Scalabilità**: Mantenimento delle prestazioni indipendentemente dalla struttura dei team

In conclusione, la relazione **Registrazione** risulta **vantaggiosa** poichè:

1. **Riduzione drastica degli accessi**: da 7400 a 3900 accessi/giorno (-47,3%)
2. **Costo di memoria giustificato**: 12 KB per un risparmio di 3500 accessi/giorno

La relazione **Registrazione** rappresenta quindi una **scelta progettuale fondamentale** e **non una ridondanza dannosa** poiché:

$$\text{Benefici}_{\text{prestazioni}} + \text{Benefici}_{\text{architetturali}} \gg \text{Costi}_{\text{memoria}} \tag{49}$$

Nota

Tutte le stime e i valori numerici presentati nella presente analisi derivano da assunzioni basate su scenari possibili dell'utilizzo della piattaforma per la gestione di hackathon. I parametri di frequenza delle operazioni, il numero di entità coinvolte e le dimensioni dei dati costituiscono una base di riferimento per la valutazione comparativa delle prestazioni e possono essere adattate in funzione delle specifiche caratteristiche operative dell'implementazione finale.

2.8.2 Eliminazione delle Generalizzazioni

Poiché i sistemi tradizionali per la gestione di basi di dati non consentono di rappresentare direttamente una generalizzazione, è necessario tradurle usando altri costrutti della modellazione concettuale. Le opzioni possibili sono le seguenti:

1. **Accorpamento delle figlie della generalizzazione nell'entità padre**
   Questa tecnica consiste nel **trasportare** attributi e relative associazioni presenti nelle entità figlie all'interno dell'entità padre. Gli eventuali attributi devono poter presentare **valore nullo** e le eventuali relazioni devono esibire un **vincolo di partecipazione parziale**, per poter rappresentare appieno tutte le entità soppresse.
2. **Accorpamento del padre della generalizzazione nelle entità figlie**
   La seconda tecnica è il *diretto opposto* della prima, nella quale si procede **inglobando** l'entità generale nelle sue specializzazioni. I suoi attributi si distribuiscono tra le figlie e le sue associazioni si moltiplicano per collegarsi alle entità restanti, preservando gli **stessi vincoli di cardinalità**.
3. **Sostituzione della generalizzazione con associazioni**
   Nell'ultima tecnica si procede invece preservando tutte le entità coinvolte e **introducendo delle associazioni** tra il padre e ogni sua figlia, conservando quelle già presenti. Queste nuove associazioni — dette *is-a* — sono di tipo **1:1 con vincolo di partecipazione parziale** verso il padre; questo rappresenta correttamente l'essenza di una generalizzazione, dove ogni figlia è necessariamente in relazione con il padre ma non viceversa.
   È bene sottolineare che le entità specializzate andrebbero in questa fase descritte come **deboli** e identificate tramite le relazioni appena create, ma il successivo processo di identificazione di chiavi principali consente di sorvolare questo dettaglio.

Una volta descritte le diverse possibilità di ristrutturazione della gerarchia, si illustrano adesso i **motivi dietro alla scelta** di una di esse come segue:

Alternativa 1	**Alternativa 2**	**Alternativa 3**
Questa opzione è da **evitare** in quanto, assegnando tutte le operazioni dei vari utenti — che differiscono significativamente — ad una *singola entità*, introdurremmo un'ammontare di **valori nulli** eccessiva, oltre a complicare in modo non banale la chiarezza del sistema.	Questa tecnica è invece **inattuabile** poiché la generalizzazione presenta un *vincolo di completezza parziale*; ciò significa che accorpare **Utente** nelle entità figlie porterebbe ad una perdita di rappresentazione per quelle istanze di **Utente** che non cadono in nessuna delle tre specializzazioni.	La soluzione ritenuta **pertinente** è stata quindi la presente in quanto garantisce assenza di valori nulli nelle entità e la corretta rappresentazione del mini-world, al costo di un numero di accessi aggiuntivi che non impatta gravosamente sulle prestazioni della base di dati.

2.8.3 Eliminazione Attributi Multivalore

Durante la progettazione conceuttuale non sono stati individuati attributi multivalore da elidere eventualmente, pertanto si procede con la fase successiva.

2.8.4 Eliminazione Attributi Strutturati

Poichè non sono presenti attributi strutturati da scomporre, è stato quindi possibile procedere con l'ultima fase della ristrutturazione.

2.8.5 Partizionamento/Accorpamento di Entità e Relazioni

Le entità e associazioni individuate non necessitano di essere modificate ulteriormente tramite partizionamento o accorpamento, segue quindi l'analisi dell'ultima fase della ristrutturazione del modello concettuale.

2.8.6 Scelta degli Identificatori Principali

In questa sezione si illustra la scelta delle chiavi candidate per la successiva introduzione delle chiavi primarie nel sistema:

- **Utente:** l'attributo *Username* è stato ritenuto adatto per essere un attributo identificatore poichè tramite quest'ultimo è possibile distinguere univocamente ogni utente.
- **Partecipante — Organizzatore — Giudice**: in quanto specializzazioni dell'entità Utente, faranno riferimento all'attributo identificatore *Username* della generalizzazione **Utente**. Per rendere più forte il vincolo di unicità, introduciamo quindi un attributo `ID` per Partecipante, Organizzatore e Giudice per identificare la precisa istanza specializzata, permettendo inoltre di preservare la semplicità del sistema.
- **Team**: Durante la presente fase di scelta degli identificatori, un attributo di dominio già presente ha rappresentato un potenziale candidato:
  - l'attributo *Nome* avrebbe costituito un'opzione valida in quanto nel contesto di un singolo Hackathon non possono esistere due Team con lo stesso nome. D'altra parte considerando la base di dati nel suo insieme tale attributo non impone un vincolo di unicità tanto forte tale da poter identificare un singolo Team.

  È quindi opportuno introdurre un nuovo attributo identificatore "`idTeam`".
- **Documento:** L'entità Documento non presenta attributi tali da poter costituire un identificatore per via dello stretto legame con l'entità **Team**: pertanto è stato necessario introdurre un attributo che potesse invece rivestire quest'ultimo ruolo adeguatamente: "`idDocumento`".
- Nell'entità **Hackathon**, l'identificazione univoca delle istanze presenta una problematica progettuale che richiede un'analisi formale delle alternative disponibili:
  Dal punto di vista teorico, l'identificazione naturale di un hackathon può essere espressa attraverso la chiave candidata composta:

  ```
  (Titolo, Sede)
  ```

  Questa scelta si basa sulla considerazione secondo cui l'attributo *Titolo* da solo non garantisce l'unicità, poiché nella presente progettazione concettuale è stato considerato oppurtuno prevedere scenari in cui hackathon con denominazioni identiche possano svolgersi in contesti geografici diversi.
  La combinazione *Titolo* × *Sede* fornisce invece un identificatore semanticamente significativo ed univoco.
  Tuttavia, l'adozione di una chiave primaria composta introduce significative **complessità operative** nell'implementazione del sistema:
  - **Propagazione referenziale**: ogni relazione che referenzia *Hackathon* deve trasportare entrambi gli attributi come chiave esterna composta.
    È stata quindi adottata una soluzione di compromesso basata sull'introduzione di una chiave surrogata "`idHackathon`" mantenendo contemporaneamente il vincolo di unicità sulla chiave naturale:

    ```
    UNIQUE(Titolo, Sede)
    ```
    (50)

    Facendo in questo modo è preservata la semantica del dominio attraverso il vincolo di unicità naturale, mentre la chiave surrogata garantisce:
    - Semplicità nelle relazioni referenziali (singolo attributo intero)

2.8.7 Diagramma Concettuale Ristrutturato — ER

Definizione: Un **diagramma ristrutturato** è il risultato della sottoposizione del modello concettuale al processo di ristrutturazione. Esso mantiene invariato il contenuto informativo dello schema originale ma pone le basi strutturali per la successiva traduzione dello stesso nello schema logico.

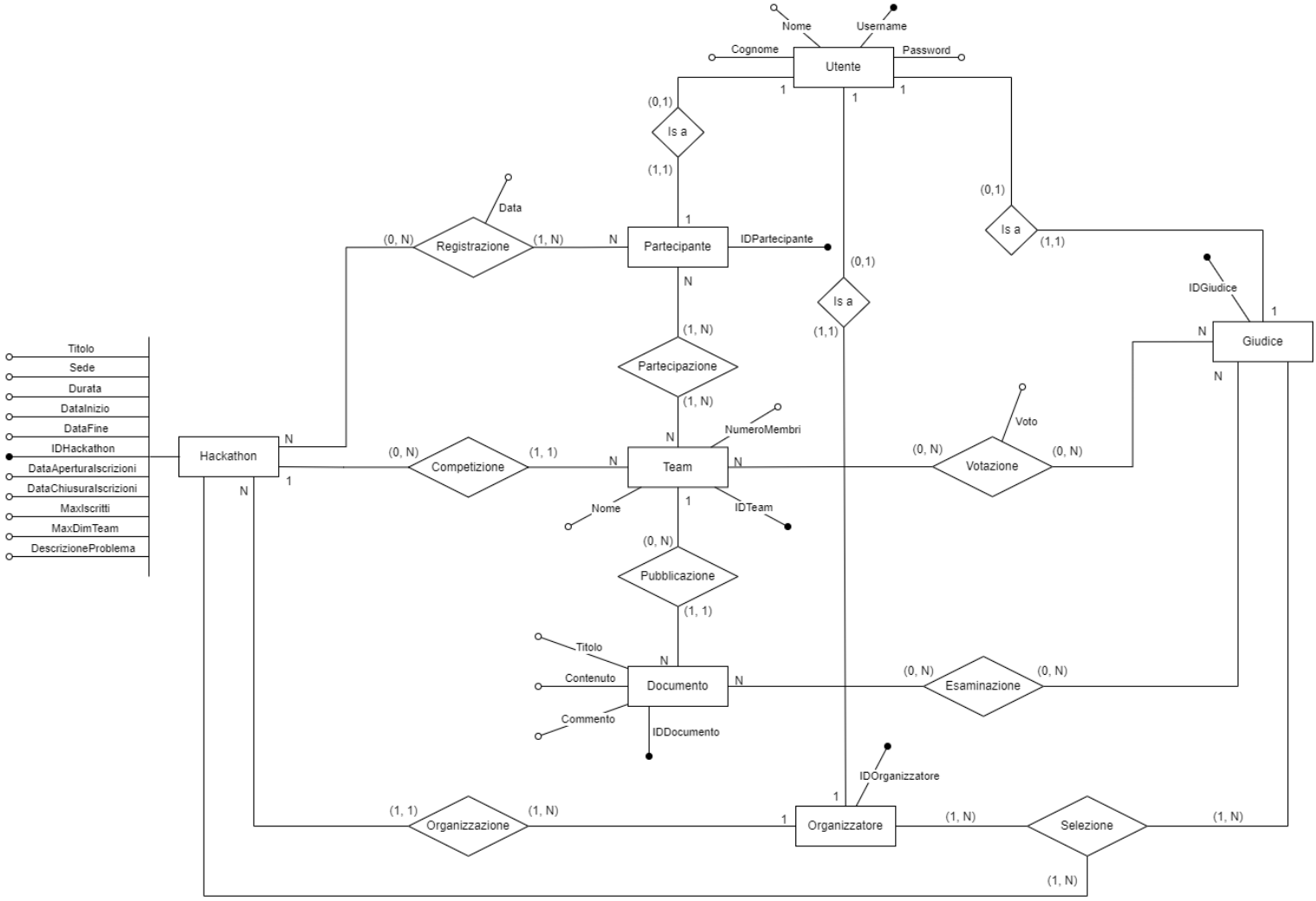
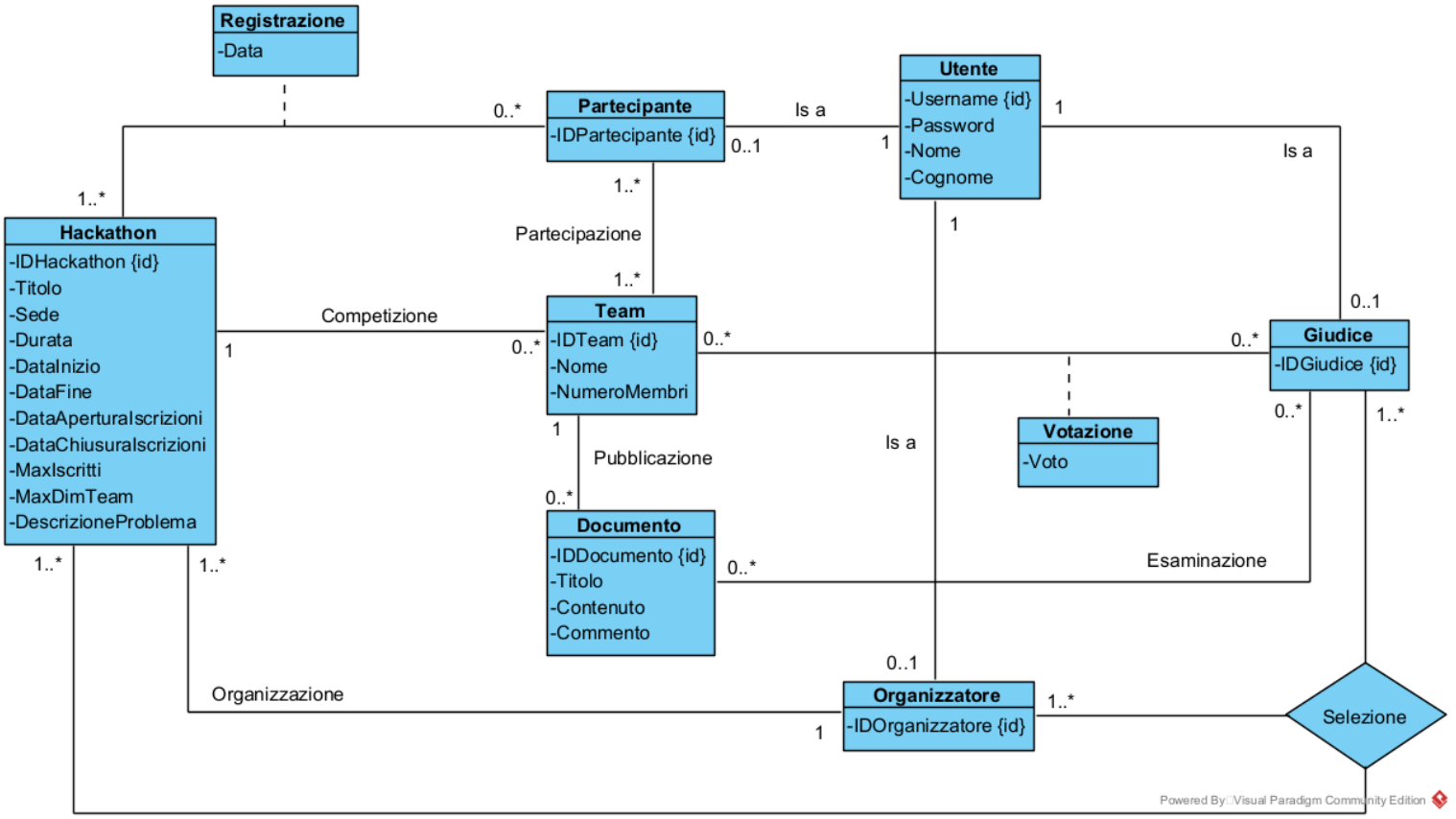


Figura 2: Stereotipizzazione Atzeni, Ceri, Paraboschi, Fraternali, Torlone

2.8.8 Diagramma Concettuale Ristrutturato — UML

Diagramma ristrutturato espresso tramite i formalismi dell'UML.



Convenzioni Notazionali

Attributo {id} → Attributo **Chiave Primaria**

2.8.9 Dizionario delle Entità

Entità	Attributi	Descrizione
Utente	- Username - Password - Nome - Cognome	Persona iscritta alla piattaforma che può prendere parte agli eventi disponibili
Partecipante	- idPartecipante	Membro di un **Team** che partecipa all'**Hackathon**
Giudice	- idGiudice	**Utente** selezionato che può giudicare e valutare gli elaborati delle squadre
Organizzatore	- idOrganizzatore	Definisce le informazioni necessarie per pubblicare un **Hackathon**
Team	- idTeam - Nome - NumeroMembri	Squadra di **Partecipanti** che ha il compito di pubblicare i progressi durante la competizione
Documento	- idDocumento - Titolo - Contenuto - Commento	File prodotto dai vari **Team** che verrà poi valutato dai **Giudici**
Hackathon	- idHackathon - Titolo - Sede - Durata - DataInizio - DataFine - DataAperturaIscrizioni - DataChiusuraIscrizioni - MaxPartecipanti - DimMaxTeam - DescrizioneProblema	Competizione istituita da un **Organizzatore**

Tabella 3: Descrizione delle entità del sistema dopo la **ristrutturazione**

2.8.10 Dizionario delle Relazioni

Relazioni	Tipologia	Descrizione
Competizione	Uno-a-Molti	- Un Team compete in un 1 singolo Hackathon - In 1 Hackathon competono nessuno o più Team
Pubblicazione	Uno-a-Molti	- Un Team pubblica nessuno o più Documenti - Un Documento è pubblicato da 1 ed un solo Team
Esaminazione	Molti-a-Molti	- Un Giudice esamina nessuno o più Documenti - Un Documento è esaminato da nessuno o più Giudici
Organizzazione	Uno-a-Molti	- Un Organizzatore organizza 1 o più Hackathon - Un Hackathon è organizzato da 1 Organizzatore
Selezione	Molti-a-Molti-a-Molti	- Un Organizzatore seleziona 2 o più Giudici nel contesto di 1 o più Hackathon - Un Giudice è selezionato da 1 o più Organizzatori nel contesto di 1 o più Hackathon - In un Hackathon sono selezionati 2 o più Giudici da 1 Organizzatore
Partecipazione	Molti-a-Molti	- Un Partecipante partecipa ad 1 o più Team - Un Team possiede 1 o più Partecipanti
Registrazione	Molti-a-Molti	- **Data**: attributo di relazione per memorizzare la data di Registrazione all'Hackathon - Un Partecipante si registra a 1 o più Hackathon - Ad un Hackathon si registrano nessuno o più Partecipanti
Votazione	Molti-a-Molti	- **Voto**: attributo di relazione che individua il valore numerico della votazione - Un Team è valutato da nessuno o più Giudici - Un Giudice valuta nessuno o più Team
Is a (Partecipante)	Uno-a-Uno	Un Utente può essere un Partecipante; Un Partecipante è un Utente
Is a (Organizzatore)	Uno-a-Uno	Un Utente può essere un Organizzatore; Un Organizzatore è un Utente
Is a (Giudice)	Uno-a-Uno	Un Utente può essere un Giudice; Un Giudice è un Utente

Tabella 4: Descrizione delle relazioni tra le entità individuate

Convenzioni Notazionali

Nome Relazione → Relazione derivante dalla Ristrutturazione della gerarchia individuata

2.8.11 Dizionario dei Vincoli

#	Nome Vincolo	Tipologia	Descrizione
1	Durata positiva	Intra-Relazionale	La durata di un **Hackathon** deve essere strettamente maggiore di 0.
2	Date corrette	Intra-Relazionale	Le date di apertura, chiusura iscrizioni, inizio e fine devono rispettare l'ordine logico: apertura $<$ chiusura iscrizioni $\leq$ inizio $<$ fine.
3	Limite inferiore per *MaxIscritti*	Intra-Relazionale	Un **Hackathon** deve consentire l'iscrizione ad almeno 2 **Partecipanti**.
4	Limite inferiore per *MaxDimTeam*	Intra-Relazionale	Ogni **Team** deve consentire la presenza di un numero di membri $>$ di 0.
5	Vincolo di unicità per la coppia *(Titolo, Sede)*	Intra-Relazionale	La coppia di attributi *(Titolo, Sede)* è unica per ogni **Hackathon**.
6	Membri di un Team positivi	Intra-Relazionale	Il numero di membri in un **Team** deve essere > 0.
7	Dominio del voto	Intra-Relazionale	Il *voto* finale deve essere compreso tra 0 e 10 estremi inclusi.
8	Vincolo di disgiunzione intra-hackathon	Inter-Relazionale	Nel contesto di un singolo **Hackathon**, un **Partecipante** non può ricoprire un secondo ruolo.
9	Team Minimi per Hackathon	Inter-Relazionale	Un **Hackathon**, per essere considerato valido, deve prevedere almeno 2 **Team**.
10	Giudici Minimi *	Inter-Relazionale	Un gruppo di **Giudici** per un **Hackathon** deve essere composto da almeno 2 membri.
11	Vincolo di unicità per l'attributo *Nome*	Inter-Relazionale	L'attributo *Nome* di un **Team** deve essere univoco per ogni **Hackathon**.

Tabella 5: Per i vincoli indicati con * vedasi la sezione Assiomi per la Progettazione Fisica

2.8.11 Dizionario dei Vincoli — Continuazione

#	Nome Vincolo	Tipologia	Descrizione
12	Completezza di esaminazione	Inter-Relazionale	Prima di inserire una *Votazione* per un **Team**, un **Giudice** deve aver esaminato prima tutti i relativi documenti.
13	Vincolo di unicità per gli attributi *Titolo* e *Contenuto*	Inter-Relazionale	Non devono esistere duplicazioni degli attributi *Titolo* e *Contenuto* di un documento contemporaneamente nel contesto di un hackathon.
14	Correttezza dell'iscrizione	Inter-Relazionale	Un **Partecipante** può iscriversi ad un **Hackathon** solo nella finestra temporale disponibile
15	Correttezza dell'inizio competizione	Intra-Relazionale	Un **Hackathon** non può essere avviato in date diverse da quelle indicate nel relativo record.
16	Esistenza di un progresso	Inter-Relazionale	Un **Team** deve aver pubblicato almeno un progresso nella competizione di riferimento prima di poter ricevere una votazione.
17	Correttezza del cambio team	Inter-Relazionale	Per poter cambiare **Team**, un **Partecipante** deve rispettare i vincoli temporali e i limiti di dimensione imposti dall'**Organizzatore**.
18	Completezza delle votazioni	Inter-Relazionale	Prima di poter produrre la tabella dei vincitori, è necessario che ogni **Team** presenti tutti i voti necessari.

Tabella 6: Vincoli del Sistema

3 Progettazione Logica

Introduzione

> *Definizione*: La progettazione logica è il processo di trasformazione dello schema concettuale in uno **schema logico** che rappresenta la struttura dei dati secondo un particolare modello logico dei dati.

Per la presente piattaforma il modello logico scelto è stato il **modello relazionale**. Nel contesto di utilizzo di tale la progettazione logica comporta:

- **Per ogni entità nel modello concettuale**: uno schema di relazione con lo stesso nome avente
  - i medesimi attributi dell'entità
  - per chiave il suo identificatore
- **Per ogni relazione nel modello concettuale**: uno schema di relazione con lo stesso nome avente
  - per attributi: gli attributi della relazione
  - per chiave: gli identificatori delle entità coinvolte

  Tutto ciò distinguendo i diversi casi in base ai vincoli di partecipazione.

È quindi necessario un processo di traduzione sistematica tale che, a partire dalle entità e dalle relazioni individuate nel modello concettuale, permetta di definirne formalmente i corrispondenti nel modello logico relazionale. Il suddetto processo è quello del **mapping logico-relazionale**.

3.1 Mapping Logico-Relazionale

Di seguito vengono dettagliate le regole formali di traduzione applicate a ciascuna entità e relazione del modello concettuale:

3.1.1 Entità

`Utente` Mappata nella relazione:

`Utente (username, password, nome, cognome)`

`Giudice` Mappata nella relazione:

`Giudice (id_giudice)`

`Documento` Mappata nella relazione:

`Documento (id_documento, commento, contenuto, titolo)`

`Team` Mappata nella relazione:

`Team (id_team, nome, numero_membri)`

`Partecipante` Mappata nella relazione:

`Partecipante (id_partecipante)`

`Hackathon` Mappata nella relazione:

`Hackathon (id_hackathon, titolo, sede, durata, data_inizio, data_fine, descrizione_problema, data_apertura_iscrizioni, data_chiusura_iscrizioni, max_iscritti, max_dim_team)`

`Organizzatore` Mappata nella relazione:

`Organizzatore (id_organizzatore)`

3.1.2 Relazione Molti-a-Molti-a-Molti (M:N:P)

Relazione *Selezione*

Tabelle coinvolte:

- `Giudice`
- `Organizzatore`
- `Hackathon`

Mapping: Viene introdotta una nuova relazione:

`Selezione (id_giudice, id_organizzatore, id_hackathon)`

provvista degli attributi chiave primaria delle tabelle coinvolte.

3.1.3 Relazioni Molti-a-Molti (N:N)

Relazione *Esaminazione*

Tabelle coinvolte:

- `Giudice`
- `Documento`

Mapping: Viene introdotta una nuova relazione:

`Esaminazione (id_giudice, id_documento)`

provvista degli attributi chiave primaria delle entità coinvolte.

Relazione *Votazione*

Tabelle coinvolte:

- `Team`
- `Giudice`

Mapping: Viene introdotta una nuova relazione:

`Votazione (id_giudice, id_team, voto)`

provvista degli attributi chiave primaria delle entità coinvolte e dell'attributo descrittivo `voto`.

Relazione *Partecipazione*

Tabelle coinvolte:

- `Partecipante`
- `Team`

Mapping: Viene introdotta una nuova relazione:

`Partecipazione (id_team, id_partecipante)`

provvista degli attributi chiave primaria delle entità coinvolte.

Relazione *Registrazione*

Tabelle coinvolte:

- `Hackathon`
- `Partecipante`

Mapping: Viene introdotta una nuova relazione:

`Registrazione (id_partecipante, id_hackathon, data)`

provvista degli attributi chiave primaria delle entità coinvolte e dell'attributo accessorio `data`.

3.1.4 Relazioni Uno-a-Molti (1:N)

Relazione *Organizzazione*

Tabelle coinvolte:

- `Organizzatore`
- `Hackathon`

Mapping: La chiave primaria dell'entità `Organizzatore` viene inserita come chiave esterna nella relazione `Hackathon`

Relazione *Pubblicazione*

Tabelle coinvolte:

- `Team`
- `Documento`

Mapping: La chiave primaria dell'entità `Team` viene inserita come chiave esterna nella relazione `Documento`

Relazione *Competizione*

Tabelle coinvolte:

- `Hackathon`
- `Team`

Mapping: La chiave primaria dell'entità `Hackathon` viene inserita come chiave esterna nella relazione `Team`

3.1.5 Relazioni Uno-a-Uno (1:1)

Relazione *Is a — Partecipante*

Tabelle coinvolte:

- `Utente`
- `Partecipante`

Mapping: La chiave primaria dell'entità `Utente` viene inserita come chiave esterna nella relazione `Partecipante`

Relazione *Is a — Organizzatore*

Tabelle coinvolte:

- `Utente`
- `Organizzatore`

Mapping: La chiave primaria dell'entità `Utente` viene inserita come chiave esterna nella relazione `Organizzatore`

Relazione *Is a — Giudice*

Tabelle coinvolte:

- `Utente`
- `Giudice`

Mapping: La chiave primaria dell'entità `Utente` viene inserita come chiave esterna nella relazione `Giudice`

3.1.6 Schema Logico-Relazionale

Il seguente schema logico-relazionale rappresenta il **risultato** del processo di **mapping logico-relazionale** applicato agli elementi costituenti lo schema concettuale ristrutturato precedentemente illustrato.

Utente (username, nome, cognome, password)

Partecipante (id_partecipante, username)
Partecipante.username → Utente.username

Organizzatore (id_organizzatore, username)
Organizzatore.username → Utente.username

Giudice (id_giudice, username)
Giudice.username → Utente.username

Esaminazione (id_giudice, id_documento)
Esaminazione.id_giudice → Giudice.id_giudice
Esaminazione.id_documento → Documento.id_documento

Documento (id_documento, commento, contenuto, titolo, id_team)
Documento.id_team → Team.id_team

Team (id_team, nome, numero_membri, id_hackathon)
Team.id_hackathon → Hackathon.id_hackathon

Partecipazione (id_team, id_partecipante)
Partecipazione.id_team → Team.id_team
Partecipazione.id_partecipante → Partecipante.id_partecipante

Hackathon (id_hackathon, titolo, sede, durata, data_inizio, data_fine, descrizione_problema, data_apertura_iscrizioni, data_chiusura_iscrizioni, max_iscritti, max_dim_team, id_organizzatore)
Hackathon.id_organizzatore → Organizzatore.id_organizzatore

Votazione (id_team, id_giudice, voto)
Votazione.id_team → Team.id_team
Votazione.id_giudice → Giudice.id_giudice

Selezione (id_hackathon, id_organizzatore, id_giudice)
Selezione.id_hackathon → Hackathon.id_hackathon
Selezione.id_organizzatore → Organizzatore.id_organizzatore
Selezione.id_giudice → Giudice.id_giudice

Registrazione (id_partecipante, id_hackathon, data)
Registrazione.id_partecipante → Partecipante.id_partecipante
Registrazione.id_hackathon → Hackathon.id_hackathon

Convenzioni Notazionali

Attributo → Chiave Primaria della Relazione
Attributo → Chiave Esterna

4 Progettazione Fisica

Introduzione

Definizione: La progettazione fisica rappresenta la fase del processo di progettazione in cui si definiscono gli aspetti implementativi riguardo la memorizzazione effettiva dei dati e la gestione di questi da parte del DBMS.

4.1 Assiomi per la Progettazione Fisica

L'introduzione di assiomi rappresenta una scelta metodologica deliberata, finalizzata a stabilire i principi fondamentali che governeranno l'intero sistema informativo **prima** dell'avvio della fase di **progettazione fisica**. Nello *scope* della presente piattaforma, la definizione di assiomi evita **un'eccessiva granularità** nell'implementazione dei vincoli.
L'approccio assiomatico permette di identificare quali **vincoli** richiedano **un'implementazione procedurale complessa** e quali possano essere gestiti attraverso **meccanismi più semplici** o **delegati** ad altri livelli dell'architettura applicativa.

Le **definizioni** degli elementi di seguito menzionati si trovano alla sezione Formalizzazione dei Vincoli.

(1) Immutabilità degli Attributi di Hackathon

Sia H un hackathon e sia S lo stato di creazione della competizione. Una volta raggiunto lo stato S, tutti gli attributi configurazionali di H divengono **immutabili** per garantire la consistenza referenziale dell'intera base di dati.

Formalmente:

$$\boxed{\forall a \in \text{Attributi}(H),\ t \geq t_S : \text{Immutabile}(a, t)}$$

dove t_S è il tempo di raggiungimento dello stato S e $\text{Immutabile}(a, t)$ indica che l'attributo a non può essere modificato al tempo t.

Tale principio assicura l'integrità dei dati correlati e previene incongruenze sistemiche derivanti da modifiche post-creazione.

(2) Completezza della Votazione

La procedura di votazione da parte di un giudice è **corretta** e ha **significato** se e solo se ogni Team ha pubblicato tutti i propri documenti prima della fine dell'Hackathon.

Formalmente:

Per ogni giudice $g \in G$, la procedura di votazione è **corretta** se e solo se:

$$\boxed{\text{Votazione}(g) \ \Leftrightarrow\ \forall d \in D,\ \text{Pubblicato}(d, t_f)}$$

dove $\text{Pubblicato}(d, t_f)$ indica che il documento d è stato pubblicato entro il tempo di fine t_f dell'hackathon H.

Questo principio stabilisce una convenzione contrattuale tra team e giudici, secondo cui il processo valutativo può essere avviato unicamente a seguito della pubblicazione integrale della documentazione richiesta, e deve essere completato entro i termini dell'hackathon.

(3) Transizioni Temporali degli Stati di Hackathon da parte dell'Organizzatore

Sia H un hackathon organizzato da $o \in O$, con i seguenti stati:

- S_C : stato di **creazione**;
- S_I : stato di **inizio**;
- S_F : stato di **fine**.

L'organizzatore deve garantire che l'hackathon:

1. ($S_c \rightarrow S_i$) transiti nello stato S_i entro la data pianificata in S_c (`dataInizio`);
2. termini ($S_i \rightarrow S_f$) esattamente nella data pianificata (`dataFine`);
3. non possa tornare a stati precedenti una volta raggiunto S_I o S_F.

Formalmente:

- Quando H è nello stato di creazione (S_C), l'organizzatore o può permetterne la transizione nello stato di inizio (S_I) se e solo se `dataCorrente` = (`dataInizio`) e **non prima**.
- Quando H è nello stato di inizio (S_I), l'organizzatore o può permetterne la transizione nello stato di fine (S_F) se e solo se `dataCorrente` = (`dataFine`) **e non dopo**.
- Nessun ritorno agli stati precedenti è permesso: una volta iniziato, H non può tornare a S_C, e una volta terminato, non può tornare a S_I.

In altre parole, le transizioni ammesse sono:

$$S_C \xrightarrow{\text{dataInizio}} S_I \xrightarrow{\text{dataFine}} S_F$$

e ogni transizione deve avvenire esattamente nelle date pianificate.

4.2 Definizione delle Tabelle Fisiche

La seguente implementazione in linguaggio SQL rispetta l'ordine di creazione delle tabelle necessario per garantire l'integrità referenziale delle chiavi esterne.

```sql
-- 1. Tabella base senza dipendenze
CREATE TABLE IF NOT EXISTS Utente (
    username VARCHAR(30) PRIMARY KEY,
    nome VARCHAR(30) NOT NULL,
    cognome VARCHAR(30) NOT NULL,
    password VARCHAR(100) NOT NULL
);
```

```sql
-- 2. Tabelle che dipendono da Utente
CREATE TABLE IF NOT EXISTS Organizzatore (
    id_organizzatore SERIAL PRIMARY KEY,
    username VARCHAR(30) NOT NULL,
    FOREIGN KEY (username) REFERENCES Utente(username)
        ON UPDATE CASCADE,
        ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Partecipante (
    id_partecipante SERIAL PRIMARY KEY,
    username VARCHAR(30) NOT NULL,
    FOREIGN KEY (username) REFERENCES Utente(username)
        ON UPDATE CASCADE,
        ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Giudice (
    id_giudice SERIAL PRIMARY KEY,
    username VARCHAR(30) NOT NULL,
    FOREIGN KEY (username) REFERENCES Utente(username)
        ON UPDATE CASCADE,
        ON DELETE CASCADE
);
```

```sql
-- 3. Tabelle che dipendono da Organizzatore
CREATE TABLE IF NOT EXISTS Hackathon (
    id_hackathon SERIAL PRIMARY KEY,
    titolo VARCHAR(50) NOT NULL,
    sede VARCHAR(30) NOT NULL,
    durata INT NOT NULL,
    data_inizio DATE NOT NULL,
    data_fine DATE NOT NULL,
    descrizione_problema TEXT DEFAULT 'Descrizione assente.',
    data_apertura_iscrizioni DATE NOT NULL,
    data_chiusura_iscrizioni DATE NOT NULL,
    max_iscritti INT NOT NULL,
    max_dim_team INT NOT NULL,
    id_organizzatore INT NOT NULL DEFAULT 0,
    FOREIGN KEY (id_organizzatore) REFERENCES Organizzatore(id_organizzatore)
        ON UPDATE CASCADE,
        ON DELETE SET DEFAULT,
    CONSTRAINT durata_positiva CHECK (durata > 0),                /* [1] */
    CONSTRAINT date_corrette CHECK (                              /* [2] */
        (data_apertura_iscrizioni < data_chiusura_iscrizioni)
        AND (data_chiusura_iscrizioni <= data_inizio)
        AND (data_inizio < data_fine) ),
    CONSTRAINT max_iscritti_minimo CHECK (max_iscritti > 1),      /* [3] */
    CONSTRAINT max_dim_team_minima CHECK (max_dim_team > 0),      /* [4] */
    CONSTRAINT chiave_naturale UNIQUE (titolo, sede)              /* [5] */
);
```

```sql
-- 4. Tabelle che dipendono da Hackathon
CREATE TABLE IF NOT EXISTS Team (
    id_team SERIAL PRIMARY KEY,
    nome VARCHAR(20) NOT NULL,
    numero_membri INT NOT NULL,
    id_hackathon INT NOT NULL,
    FOREIGN KEY (id_hackathon) REFERENCES Hackathon(id_hackathon)
        ON UPDATE CASCADE,
        ON DELETE CASCADE,
    CONSTRAINT membri_positivi CHECK (numero_membri > 0)          /* [6] */
);
```

```sql
-- 5. Tabelle che dipendono da Team
CREATE TABLE IF NOT EXISTS Documento (
    id_documento SERIAL PRIMARY KEY,
    commento TEXT DEFAULT 'Commento assente.',
    contenuto TEXT NOT NULL,
    titolo VARCHAR(50) NOT NULL,
    id_team INT NOT NULL,
    FOREIGN KEY (id_team) REFERENCES Team(id_team)
        ON UPDATE CASCADE,
        ON DELETE CASCADE
);
```

```sql
-- 7. Tabelle di Relazione
CREATE TABLE IF NOT EXISTS Registrazione (
    id_partecipante INT NOT NULL DEFAULT 0,
    id_hackathon INT NOT NULL,
    data DATE NOT NULL,
    PRIMARY KEY (id_partecipante, id_hackathon),
    FOREIGN KEY (id_partecipante) REFERENCES Partecipante(id_partecipante)
        ON UPDATE CASCADE,
        ON DELETE SET DEFAULT,
    FOREIGN KEY (id_hackathon) REFERENCES Hackathon(id_hackathon)
        ON UPDATE CASCADE,
        ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Partecipazione (
    id_team INT NOT NULL,
    id_partecipante INT NOT NULL DEFAULT 0,
    PRIMARY KEY (id_team, id_partecipante),
    FOREIGN KEY (id_team) REFERENCES Team(id_team)
        ON UPDATE CASCADE,
        ON DELETE CASCADE,
    FOREIGN KEY (id_partecipante) REFERENCES Partecipante(id_partecipante)
        ON UPDATE CASCADE,
        ON DELETE SET DEFAULT
);

CREATE TABLE IF NOT EXISTS Esaminazione (
    id_giudice INT NOT NULL DEFAULT 0,
    id_documento INT NOT NULL,
    PRIMARY KEY (id_giudice, id_documento),
    FOREIGN KEY (id_giudice) REFERENCES Giudice(id_giudice)
        ON UPDATE CASCADE,
        ON DELETE SET DEFAULT,
    FOREIGN KEY (id_documento) REFERENCES Documento(id_documento)
        ON UPDATE CASCADE,
        ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Votazione (
    id_giudice INT NOT NULL DEFAULT 0,
    id_team INT NOT NULL,
    voto dominio_voto NOT NULL,
    PRIMARY KEY (id_giudice, id_team),
    FOREIGN KEY (id_giudice) REFERENCES Giudice(id_giudice)
        ON UPDATE CASCADE,
        ON DELETE SET DEFAULT,
    FOREIGN KEY (id_team) REFERENCES Team(id_team)
        ON UPDATE CASCADE,
        ON DELETE CASCADE
);
```

```sql
-- 8. Tabella di Relazione Ternaria
CREATE TABLE IF NOT EXISTS Selezione (
    id_hackathon INT NOT NULL,
    id_organizzatore INT NOT NULL DEFAULT 0,
    id_giudice INT NOT NULL DEFAULT 0,
    PRIMARY KEY (id_hackathon, id_organizzatore, id_giudice),
    FOREIGN KEY (id_hackathon) REFERENCES Hackathon(id_hackathon)
        ON UPDATE CASCADE,
        ON DELETE CASCADE,
    FOREIGN KEY (id_organizzatore) REFERENCES Organizzatore(id_organizzatore)
        ON UPDATE CASCADE,
        ON DELETE SET DEFAULT,
    FOREIGN KEY (id_giudice) REFERENCES Giudice(id_giudice)
        ON UPDATE CASCADE,
        ON DELETE SET DEFAULT
);
```

Note

La numerazione presente accanto ad ogni vincolo CONSTRAINT (es. /* [n] */) corrisponde a quella dei vincoli elencati nel Dizionario dei Vincoli precedentemente illustrato.

4.3 Implementazione dei Vincoli di Base

Come precedentemente accennato, il *modello relazionale* presenta vari **vincoli formali** che ogni istanza di relazione deve soddisfare per essere correttamente espressa:

1. **Vincoli di dominio**
   Ogni attributo A di una relazione R è associato a un dominio $Dom(A)$, ossia l'insieme dei valori ammessi. Un vincolo sui domini richiede che tutti i valori assunti dagli attributi debbano appartenere al $Dom(A)$.

2. **Vincoli di chiave**
   In una relazione R, ogni attributo di *chiave* deve essere **unico**, senza presentare mai valori duplicati.

3. **Vincoli di integrità di entità**
   Un attributo di *chiave primaria* non può presentare valore NULL per garantire l'esistenza del record.

4. **Vincoli di integrità referenziale**
   Una *chiave esterna* deve sempre riferirsi ad una chiave primaria *esistente* o essere posta a NULL per certificare la **consistenza delle informazioni**.

Questi vincoli formali assumono forma più *esplicita* tramite le seguenti varianti, che descrivono propriamente tramite il linguaggio `SQL` le condizioni necessarie per mantenere **coerenza** nella base di dati:

1. **Vincoli sui domini degli attributi**
2. **Vincoli di chiave primaria**
3. **Vincoli di integrità referenziale**
4. **Vincoli sui valori NULL e valori predefiniti**
5. **Vincoli sulle tuple**

1. Vincoli sui domini degli attributi

Un vincolo sul dominio impone che un dato attributo A cada in uno specifico $Dom(A)$ determinato secondo le necessità evidenziate. Si consideri per esempio l'attributo *Voto*, il suo valore deve essere contenuto nell'internvallo [0, 10]. Si è quindi introdotto il `CONSTRAINT` *dominioVoto* per garantire che i valori inseriti rispettino il vincolo individuato.

```
-- 6. Domini per gli attributi
CREATE DOMAIN dominio_voto AS INT
    CONSTRAINT range CHECK (VALUE BETWEEN 0 AND 10)                /* [7] */
;
```

2. Vincoli di chiave primaria

Una chiave primaria è un insieme di attributi K che identifica univocamente ogni tupla di una relazione R. I vincoli associati a tale concetto impongono:

- **Unicità:** Non possono essere presenti due tuple con lo stesso valore di chiave primaria.
- **Non nullità:** Nessun attributo della chiave primaria può assumere valore NULL.

Tali vincoli sono stati implementati tramite l'utilizzo dell'apposita clausola `PRIMARY KEY`, la quale è capace di assicurare le suddette proprietà caratteristiche di una chiave primaria.
È possibile inoltre definire dei **vincoli di chiave alternativa** tramite la keyword `UNIQUE`, che permette di specificare che l'attributo (o gli attributi) devono presentare valore univoco.

3. Vincoli di integrità referenziale

L'integrità referenziale assicura che le relazioni tra tabelle siano coerenti tramite chiavi esterne. Formalmente: se R_1 ha una chiave primaria K e R_2 ha una chiave esterna FK che fa riferimento a K, allora ogni valore di FK in R_2 deve esistere come valore corrispondente di K in R_1, oppure può essere NULL se permesso.

Per ogni attributo di una tabella coinvolta in una relazione che rappresenta una **chiave esterna** (`FOREIGN KEY`) ne è stato indicato il corrispondente (`REFERENCES`). Quest'ultimo in particolare rappresenta l'attributo **chiave primaria** dell'altra entità coinvolta nella medesima relazione.

4. Vincoli sui valori NULL e valori predefiniti

L'inserimento di `NOT NULL` nella creazione di una tabella serve a definire un vincolo che **impedisce l'inserimento di valori nulli** in specifiche colonne.
Di seguito le motivazioni che hanno portato alla specifica di tale clausola con relativi esempi di contesti d'implementazione:

- **Garantire l'integrità dei dati**
  Alcune informazioni sono essenziali per il funzionamento dell'applicazione. Ad esempio, nel presente schema:
  - Il titolo di un **Hackathon** deve sempre esistere
  - Il nome di un **Team** deve essere sempre specificato
- **Evitare errori logici**
  Omettendo la clausola `NOT NULL`, potrebbero verificarsi scenari inconsistenti, dove un attributo che necessita di avere obbligatoriamente un valore è invece sprovvisto di esso.

L'inserimento di `DEFAULT` invece consente di specificare un **valore predefinito** per un attributo, in sostituzione di un dato non inserito o *eliminato*. In particolare, la seconda possibilità è stata introdotta nel sistema per garantire l'**integrità** e la **coerenza** dei dati.

Nel contesto in cui un **Utente** decide di eliminare i propri dati dalla piattaforma, si ricadrebbe in uno stato incoerente della base di dati se procedessimo all'eliminazione di tutti i riferimenti ad esso (ovverosia la clausola `ON DELETE CASCADE`). È quindi stato introdotto un **utente fantoccio** per ovviare al problema della coerenza interna tra informazioni, ricollegando ogni **Utente** non più presente nel sistema a questo nuovo record fittizio.

5. Vincoli sulle tuple

I vincoli sulle tuple sono condizioni che devono essere soddisfatte da ogni singola tupla di una relazione. Questi vincoli possono coinvolgere uno o più attributi della stessa tupla.
Un esempio di vincolo individuato appartenente a tale categoria corrisponde al `CONSTRAINT` denominato `dateCorrette` ([2]), il quale verifica per ogni tupla la relazione che intercorre tra i valori degli attributi *data_apertura_iscrizioni*, *data_chiusura_iscrizioni*, *data_fine*, *data_inizio* della tabella *Hackathon*, verificando che essi siano coerenti con la semantica che descrivono.

Dettagli implementativi e considerazioni

- I campi `SERIAL` garantiscono la generazione automatica di chiavi primarie progressive per gli identificatori numerici.
- L'ordine di inserimento dei dati deve seguire la stessa sequenza gerarchica utilizzata per la creazione delle tabelle, per assicurarsi di rispettare le **dipendenze** presenti nel sistema.

4.4 Implementazione di Altri Vincoli — Triggers

Introduzione

La restante parte di vincoli individuati deve essere necessariamente implementata tramite un costrutto più potente di un'espressione booleana o di una clausola durante la creazione delle tabelle: i **trigger**.

> *Definizione*: un `trigger` è un tipo di funzione conservata nel DBMS e innescata da uno specifico evento definito dal trigger stesso, grazie al quale è possibile automatizzare processi di controllo e verifica dei dati nel sistema.

Un esempio di vincoli la cui implementazione richiede l'utilizzo di un trigger sono i **vincoli interrelazionali** per i quali è necessario gestire logiche complesse e operazioni automatiche che richiedono l'esecuzione di codice in risposta a eventi DML su più tabelle.

Implementazione dei trigger

4.4.1 unique_team_name

Questo trigger si occupa di assicurare che in una data competizione un team non abbia lo stesso nome di un altro. Questo rispetta il vincolo di chiave naturale ([11]) locale rispetto ad un hackathon, presente nelle analisi precedenti.

```
CREATE OR REPLACE FUNCTION unique_team_name_f()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
    -- Controlla se esiste gia' un team con lo stesso nome
    IF EXISTS (SELECT 1
               FROM Team
               WHERE nome = NEW.nome AND id_hackathon = NEW.id_hackathon)
        THEN
        RAISE EXCEPTION 'Un team con il nome "%" e'' gia'' esistente', NEW.nome;
    END IF;

    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER unique_team_name_t
    BEFORE INSERT OR UPDATE OF nome ON Team
    FOR EACH ROW
    EXECUTE FUNCTION unique_team_name_f();
```

4.4.2 double_role

Questo trigger implementa il vincolo [8], garantendo che prima dell'inserimento di una nuova registrazione di un partecipante, esso non stia già ricoprendo un altro ruolo. In particolare, questo è l'unico controllo necessario, poiché una simile verifica a livello database su *Organizzatore* o *Giudice* sarebbe superflua (come spiegato nella sezione 4.1).

```
CREATE OR REPLACE FUNCTION double_role_f()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
    user_to_check Utente.username%TYPE;
    org_user      Utente.username%TYPE;
    judge_user    Utente.username%TYPE;

    judges_cursor CURSOR FOR (SELECT username
                              FROM Selezione NATURAL JOIN Giudice
                              WHERE id_hackathon = NEW.id_hackathon);
BEGIN
    -- Estraggo l'username
    SELECT username INTO user_to_check
    FROM Partecipante
    WHERE id_partecipante = NEW.id_partecipante;

    -- Prendo l'organizzatore
    SELECT username INTO org_user
    FROM Organizzatore NATURAL JOIN Hackathon
    WHERE id_hackathon = NEW.id_hackathon;

    -- Controllo prima se e' organizzatore
    IF org_user = user_to_check THEN
        RAISE EXCEPTION 'L''organizzatore non puo'' partecipare alla
            competizione';
    END IF;

    -- Con questo controllo ogni giudice
    OPEN judges_cursor;
    LOOP
        EXIT WHEN judges_cursor%NOTFOUND;
        FETCH judges_cursor INTO judge_user;
        IF judge_user = user_to_check THEN
           RAISE EXCEPTION 'Un giudice non puo'' partecipare alla competizione';
        END IF;
    END LOOP;
    CLOSE judges_cursor;
    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER double_role_t
    BEFORE INSERT ON Registrazione
    FOR EACH ROW
    EXECUTE FUNCTION double_role_f();
```

4.4.3 check_complete_examination

La seguente funzione trigger si occupa di verificare che un dato giudice abbia esaminato tutti i documenti di un dato team prima di poter inserire una votazione per esso (vincolo [12]).

```
CREATE OR REPLACE FUNCTION check_complete_examination_f()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
    total_docs    INT;
    examined_docs INT;
BEGIN
    -- Numero totale di documenti del team
    SELECT COUNT(id_documento) INTO total_docs
    FROM Documento
    WHERE id_team = NEW.id_team;

    -- Numero di documenti del team esaminati dal giudice
    SELECT COUNT(d.id_documento) INTO examined_docs
    FROM Documento d NATURAL JOIN Esaminazione e
    WHERE d.id_team = NEW.id_team AND e.id_giudice = NEW.id_giudice;

    -- Controllo di completezza
    IF total_docs <> examined_docs THEN
        RAISE EXCEPTION 'Non hai esaminato tutti i documenti del team';
    END IF;

    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER check_complete_examination_t
    BEFORE INSERT ON Votazione
    FOR EACH ROW
    EXECUTE FUNCTION check_complete_examination_f();
```

4.4.4 check_unique_document

Questo trigger controlla che nel contesto di un singolo hackathon un documento non presenti la stessa coppia di titolo e contenuto. Questo garantisce diversità e riconoscibilità del progresso all'interno della stessa competizione (vincolo [13]).

```
CREATE OR REPLACE FUNCTION check_unique_document_f()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
    v_title   Documento.titolo%TYPE;
    v_content Documento.contenuto%TYPE;
    v_hack    Hackathon.id_hackathon%TYPE;
    all_docs  REFCURSOR;
BEGIN
    -- Estraggo l'hackathon di riferimento
    SELECT id_hackathon INTO v_hack
    FROM Documento NATURAL JOIN Team
    WHERE id_documento = NEW.id_documento;
    -- Cursore per i dati di tutti i documenti di quell'hackathon
    OPEN all_docs FOR (SELECT titolo, contenuto
                       FROM Documento NATURAL JOIN Team
                       WHERE id_hackathon = v_hack);
    -- Se trovo coppie uguali il vincolo e' violato
    LOOP
        EXIT WHEN all_docs%NOTFOUND;
        FETCH all_docs INTO v_title, v_content;
        IF v_title = NEW.titolo AND v_content = NEW.contenuto THEN
            RAISE EXCEPTION 'Il titolo e il commento devono essere univoci in
                questa competizione';
        END IF;
    END LOOP;

    CLOSE all_docs;
    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER check_unique_document_t
    BEFORE INSERT ON Documento
    FOR EACH ROW
    EXECUTE FUNCTION check_unique_document_f();
```

4.5 Funzioni e Procedure

Definizioni:
(1) Una **procedura** è un blocco di codice non anonimo identificato da un nome il quale può essere richiamato più volte da altri moduli del programma.
(2) Una **funzione** è concettualmente simile a una procedura, ma si distingue da essa in quanto caratterizzata da un valore di ritorno. Tale valore può essere utilizzato all'interno di espressioni o istruzioni SQL, compatibilmente con le regole imposte dal sistema di gestione della base di dati.

Introduzione

L'uso congiunto di procedure e funzioni consente di ottenere un'organizzazione del codice più chiara, riutilizzabile e manutenibile. Tuttavia, per sfruttarne le potenzialità, è necessario un linguaggio che permetta di integrarli con le operazioni tipiche delle basi di dati.
Ciò è possibile solo tramite **PL/SQL** un linguaggio che permette l'introduzione dei costrutti classici della programmazione procedurale in SQL (*l.p. dichiarativo*). Estendendo così quest'ultimo infatti è stato possibile utilizzare:

- Costrutti condizionali
- Costrutti iterativi
- Gestione delle eccezioni

Implementazione delle procedure

4.5.1 start_hackathon

La seguente procedura si occupa di verificare le condizioni necessarie affinché un hackathon possa partire con successo. In particolare, verifica la validità del vincolo `[9]` oltre a controllare che la data di inizio sia corretta con quella al momento del lancio della procedura (vincolo `[15]`).
Si è scelto di eliminare direttamente l'istanza di hackathon in questione poiché sarebbe improprio preservare una competizione del quale dovremmo cambiare le informazioni più significative, a fronte di una più semplice riformazione dell'evento in sé.

```
CREATE OR REPLACE PROCEDURE start_hackathon (
    IN in_hack Hackathon.id_hackathon%TYPE)
LANGUAGE plpgsql
AS $$
DECLARE
    start_date      Hackathon.data_inizio%TYPE;
    teams_threshold INT;
BEGIN
    -- Controllo se la data e' corretta
    SELECT data_inizio INTO start_date
    FROM Hackathon
    WHERE id_hackathon = in_hack;
    IF CURRENT_DATE < start_date THEN
        RAISE EXCEPTION 'E'' troppo presto!';
    ELSIF CURRENT_DATE > start_date THEN
        CALL delete_hackathon(in_hack);
        RAISE EXCEPTION 'Hai avviato troppo tardi!';
    END IF;
    -- Controllo se ci sono almeno 2 team (vincolo [9])
    SELECT COUNT(id_team) INTO teams_threshold
    FROM Hackathon NATURAL JOIN Team
    WHERE id_hackathon = in_hack;
    IF teams_threshold < 2 THEN
        CALL delete_hackathon(in_hack);
        RAISE EXCEPTION 'Non ci sono abbastanza Team iscritti.';
    END IF;
END;
$$;
```

4.5.2 authenticate

La seguente procedura consente di identificare il ruolo di un **Utente** all'interno dell'hackathon specificato come parametro di input, così da limitare le operazioni sensibili — come la pubblicazione della *descrizione_problema* — ai soli utenti autorizzati. Utilizza due parametri di output per restituire il ruolo di quell'utente e l'eventuale ID ad esso associato.

```
CREATE OR REPLACE PROCEDURE authenticate (
    IN in_hack     Hackathon.id_hackathon%TYPE,
    IN in_username Utente.username%TYPE,
    OUT o_role     INT,
    OUT o_id       INT)
LANGUAGE plpgsql
AS $$
BEGIN
    o_role := 0;
    o_id := 0;
    -- ^ Valori di default
    SELECT id_giudice INTO o_id
    FROM Selezione NATURAL JOIN Giudice
    WHERE id_hackathon = in_hack AND username = in_username;
    IF FOUND THEN
        o_role := 1;
    END IF;
    IF o_id IS NULL THEN
        SELECT id_organizzatore INTO o_id
        FROM Hackathon NATURAL JOIN Organizzatore
        WHERE id_hackathon = in_hack AND username = in_username;
        IF FOUND THEN
            o_role := 2;
        END IF;
    END IF;
    IF o_id IS NULL THEN
        SELECT id_partecipante INTO o_id
        FROM Registrazione NATURAL JOIN Partecipante
        WHERE id_hackathon = in_hack AND username = in_username;
        IF FOUND THEN
            o_role := 3;
        END IF;
    END IF;
    IF o_role = 0 THEN
        RAISE EXCEPTION 'Non fai parte di questa competizione.';
    END IF;
END;
$$;
```

4.5.3 new_user

Procedura per l'inserimento in piattaforma di un nuovo utente.

```
CREATE OR REPLACE PROCEDURE new_user (
    IN v_username Utente.username%TYPE,
    IN v_nome     Utente.nome%TYPE,
    IN v_cognome  Utente.cognome%TYPE,
    IN v_password Utente.password%TYPE)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO Utente VALUES (v_username, v_nome, v_cognome, v_password);
EXCEPTION
    WHEN unique_violation THEN -- Propago l'errore con un messaggio esplicativo
        RAISE EXCEPTION 'Questo nome utente non e'' disponibile';
END;
$$;
```

4.5.4 subscribe

La seguente procedura consente agli utenti di prendere parte agli hackathon scelti, verificando la conformità della data attuale con quelle imposte dalla competizione (vincolo [14]) ed eventualmente creando un "account" da partecipante per concedere l'accesso a quel ruolo per l'utente interessato. Viene inoltre creato un team di default nel quale il nuovo partecipante viene inserito automaticamente.

```
1  CREATE OR REPLACE PROCEDURE subscribe (
2      IN in_hack     Hackathon.id_hackathon%TYPE,
3      IN in_username Utente.username%TYPE)
4  LANGUAGE plpgsql
5  AS $$
6  DECLARE
7      start_sub_date     Hackathon.data_apertura_iscrizioni%TYPE;
8      end_sub_date       Hackathon.data_chiusura_iscrizioni%TYPE;
9      in_id_partecipante Partecipante.id_partecipante%TYPE;
10     max_subs           Hackathon.max_iscritti%TYPE;
11     actual_subs        INT;
12     new_team           Team.id_team%TYPE;
13     name               Team.nome%TYPE;
14 BEGIN
15     -- Verifico che la data attuale sia coerente con le date da rispettare
16     SELECT data_apertura_iscrizioni INTO start_sub_date
17     FROM Hackathon
18     WHERE id_hackathon = in_hack;
19     IF CURRENT_DATE < start_date THEN
20        RAISE EXCEPTION 'Non e'' ancora possibile iscriversi a questo hackathon';
21     END IF;
22     SELECT data_chiusura_iscrizioni INTO end_sub_date
23     FROM Hackathon
24     WHERE id_hackathon = in_hack;
25     IF CURRENT_DATE > end_sub_date THEN
26         RAISE EXCEPTION 'Non e'' piu'' possibile iscriversi a questo hackathon';
27     END IF;
28
29     -- Verifico che il partecipante esista
30     SELECT id_partecipante INTO in_id_partecipante
31     FROM Partecipante
32     WHERE username = in_username;
33     IF NOT FOUND THEN
34         INSERT INTO Partecipante (username) VALUES (in_username) RETURNING
               id_partecipante INTO in_id_partecipante;
35     END IF;
36
37     -- Verifico che il partecipante non sia gia' registrato
38     SELECT 1
39     FROM Registrazione
40     WHERE id_partecipante = in_id_partecipante AND id_hackathon = in_hack;
41     IF FOUND THEN
42         RAISE EXCEPTION 'Sei gia'' iscritto a questo hackathon';
43     END IF;
44
45     -- Verifico che sia possibile iscriversi entro i limiti predefiniti
46     SELECT max_iscritti INTO max_subs
47     FROM Hackathon
48     WHERE id_hackathon = in_hack;
49     SELECT COUNT(*) INTO actual_subs
50     FROM Registrazione
51     WHERE id_hackathon = in_hack;
52     IF (actual_subs+1) > max_subs THEN
53         RAISE EXCEPTION 'Il limite massimo di iscrizioni e'' stato raggiunto';
54     END IF;
55
56
```

```
    -- Inserisco la registrazione e il relativo team di default
    INSERT INTO Registrazione VALUES(in_id_partecipante, in_hack, CURRENT_DATE);
    name := FORMAT('Team di %L', in_username);
    INSERT INTO Team (nome, numero_membri, id_hackathon)
        VALUES (name, 1, in_hack) RETURNING id_team INTO new_team;
    INSERT INTO Partecipazione VALUES (new_team, in_id_partecipante);
END;
$$;
```

4.5.5 end_hackathon

Procedura per gestire correttamente tutte le operazioni gestionali per la corretta chiusura di un hackathon. Verifica che siano presenti votazioni per tutti i team da tutti i giudici — inserendo quelle mancanti di default — e restituisce la classifica finale.

```
CREATE OR REPLACE FUNCTION end_hackathon(
    in_hack Hackathon.id_hackathon%TYPE)
RETURNS refcursor
LANGUAGE plpgsql
AS $$
DECLARE
    winners     REFCURSOR;
    all_judges CURSOR FOR (SELECT id_giudice
                           FROM Selezione
                           WHERE id_hackathon = in_hack);
    one_judge  Giudice.id_giudice%TYPE;
    all_teams  CURSOR FOR (SELECT id_team
                           FROM Team
                           WHERE id_hackathon = in_hack);
    one_team   Team.id_team%TYPE;
BEGIN
    OPEN all_judges;
    OPEN all_teams;
    LOOP -- Controllo che siano presenti tutte le votazioni necessarie
        EXIT WHEN all_judges%NOTFOUND;
        FETCH all_judges INTO one_judge;
        LOOP
            EXIT WHEN all_teams%NOTFOUND;
            FETCH all_teams INTO one_team;
            CALL check_complete_grading(one_judge, one_team);
        END LOOP;
    END LOOP;
    -- Produco la classifica finale
    SELECT scoreboard (in_hack) INTO winners;
    CLOSE all_judges;
    CLOSE all_teams;
    RETURN winners;
END;
$$;
```

4.5.6 delete_hackathon

Procedura per l'eliminazione di un dato hackathon.

```
CREATE OR REPLACE PROCEDURE delete_hackathon (
    IN id_hack_to_del Hackathon.id_hackathon%TYPE)
LANGUAGE plpgsql
AS $$
BEGIN
    DELETE FROM Hackathon WHERE id_hackathon = id_hack_to_del;
END;
$$;
```

4.5.7 atleast_1_progress

Tale procedura è utile a verificare che al termine di un hackathon ogni team abbia pubblicato almeno un progresso (vincolo [16]). In caso contrario, verrà assegnato automaticamente un voto pari a zero e inserito il relativo record nella tabella `Votazione`.
Tale convenzione è stata adattata per rispettare l'

```
CREATE OR REPLACE PROCEDURE atleast_1_progress (
    IN in_hack Hackathon.id_hackathon%TYPE)
LANGUAGE plpgsql
AS $$
DECLARE
    -- Cursore per tutti i team dell'hackathon interessato
    teams_cursor  CURSOR FOR (SELECT id_team
                              FROM Team
                              WHERE id_hackathon = in_hack);
    current_team  Team.id_team%TYPE;
    all_judges    REFCURSOR;
    current_judge Giudice.id_giudice%TYPE;
BEGIN
    OPEN teams_cursor;
    -- Controllo se tutti i team hanno almeno un documento pubblicato
    LOOP
        EXIT WHEN teams_cursor%NOTFOUND;
        FETCH teams_cursor INTO current_team;
        IF current_team NOT IN (SELECT id_team
                                FROM Documento) THEN
            -- Se non c'e' nemmeno un documento allora la votazione e'
                automaticamente pari a 0
            OPEN all_judges FOR (SELECT id_giudice
                                 FROM Selezione
                                 WHERE id_hackathon = in_hack);
            LOOP
                EXIT WHEN all_judges%NOTFOUND;
                FETCH all_judges INTO current_judge;
                INSERT INTO Votazione VALUES (current_judge, current_team, 0);
            END LOOP;
            CLOSE all_judges;
        END IF;
    END LOOP;
    CLOSE teams_cursor;
END;
$$;
```

4.5.8 publish_problem

Questa procedura permette a uno dei giudici di pubblicare la descrizione al problema da affrontare nell'hackathon di cui sono esaminatori.

```
CREATE OR REPLACE PROCEDURE publish_problem (
    in_id_hackathon Hackathon.id_hackathon%TYPE,
    in_descrizione  Hackathon.descrizione_problema%TYPE)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Aggiorna la descrizione del problema
    UPDATE Hackathon
    SET descrizione_problema = in_descrizione
    WHERE id_hackathon = in_id_hackathon;
END;
$$;
```

4.5.9 join_team

Questa procedura si occupa di gestire l'entrata di un partecipate in un team diverso da quello assegnatogli di default. In particolare si verificano in primis i casi base che possono produrre errori logici, come il tentativo di partecipazione al team di cui si fa già parte. Successivamente la procedura si occupa di verificare che sia effettivamente possibile unirsi al team controllando la coerenza con la capienza massima espressa dall'attributo `max_dim_team` (vincolo [17]). Una volta garantite tutte le ipotesi di partecipazione, si procede alla sostituzione dei record vecchio con quello nuovo, incrementando l'attributo `numero_membri` del team a cui ci si vuole unire.

```
CREATE OR REPLACE PROCEDURE join_team (
    IN in_username Partecipante.username%TYPE,
    IN in_id_team  Team.id_team%TYPE)
LANGUAGE plpgsql
AS $$
DECLARE
    in_id_partecipante Partecipante.id_partecipante%TYPE;
    already_member     INT;
    max_members        Hackathon.max_dim_team%TYPE;
    actual_members     Team.numero_membri%TYPE;
    old_members        Team.numero_membri%TYPE;
    old_team           Team.id_team%TYPE;
    v_hack             Hackathon.id_hackathon%TYPE;
    start_date         Hackathon.data_inizio%TYPE;
BEGIN
    -- Verifica che l'utente esista e sia partecipante
    SELECT id_partecipante INTO in_id_partecipante
    FROM Partecipante
    WHERE username = in_username;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Partecipante "%" non trovato', in_username;
    END IF;

    -- Verifica che non sia troppo tardi per cambiare team
    SELECT id_hackathon INTO v_hack
    FROM Team
    WHERE id_team = in_id_team;
    SELECT data_inizio INTO start_date
    FROM Hackathon
    WHERE id_hackathon = v_hack;
    IF start_date < CURRENT_DATE THEN
        RAISE EXCEPTION 'E'' troppo tardi per cambiare team!';
    END IF;

    -- Verifica che il partecipante non sia gia' nel team
    SELECT COUNT(*) INTO already_member
    FROM Partecipazione
    WHERE id_team = in_id_team AND id_partecipante = in_id_partecipante;
    IF already_member > 0 THEN
        RAISE EXCEPTION 'Partecipante "%" e'' gia'' membro del team %',
            in_username, in_id_team;
    END IF;

    -- Verifica che il team non sia gia' alla capienza massima
    SELECT max_dim_team INTO max_members
    FROM Hackathon NATURAL JOIN Team
    WHERE id_team = in_id_team;
    SELECT numero_membri INTO actual_members
    FROM Team
    WHERE id_team = in_id_team;
    IF (actual_members+1) > max_members THEN
        RAISE EXCEPTION 'Il team ha gia'' raggiunto la dimensione massima
            concessa';
    END IF;
```

```
    -- Gestisco il vecchio team prima di inserire il nuovo record
    SELECT id_team INTO old_team
    FROM Partecipazione NATURAL JOIN Team
    WHERE id_partecipante = in_id_partecipante AND id_hackathon = v_hack;
    SELECT numero_membri INTO old_members
    FROM Team
    WHERE id_team = old_team;
    IF old_members-1 = 0 THEN
        DELETE FROM Team WHERE id_team = old_team;
    ELSE
        UPDATE Team SET numero_membri = (numero_membri-1) WHERE id_team =
            old_team;
        DELETE FROM Partecipazione WHERE id_partecipante = in_id_partecipante
            AND id_team = old_team;
    END IF;

    -- Inserisce la partecipazione e incrementa il contatore
    INSERT INTO Partecipazione(id_team, id_partecipante) VALUES (in_id_team,
        in_id_partecipante);
    UPDATE Team SET numero_membri = numero_membri+1 WHERE id_team = in_id_team;
END;
$$;
```

4.5.10 grade_team

Procedura per inserire correttamente una votazione per un dato team.
Sfrutta il trigger *check_complete_examination* per verificare la coerenza della nuova operazione.

```
CREATE OR REPLACE PROCEDURE grade_team(
    in_id_giudice Giudice.id_giudice%TYPE,
    in_id_team    Team.id_team%TYPE,
    in_voto       INT)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO Votazione VALUES (in_id_giudice, in_id_team, in_voto);
END;
$$;
```

4.5.11 check_complete_grading

Questa procedura è tale da garantire che alla fine della competizione ogni team abbia le votazioni necessarie per la corretta conclusione della stessa (vincolo [18]). Il voto predefinito assegnatogli in assenza di quello dei giudici è 6.

```
CREATE OR REPLACE PROCEDURE check_complete_grading(
    in_id_giudice Giudice.id_giudice%TYPE,
    in_id_team    Team.id_team%TYPE)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Controlla se il giudice ha gia' votato il team
    IF NOT EXISTS (SELECT 1
                   FROM Votazione
                   WHERE id_giudice = in_id_giudice
                      AND id_team = in_id_team)
        THEN
        -- Inserisce voto predefinito = 6 se il giudice non ha votato il team
        INSERT INTO Votazione(id_giudice, id_team, voto) VALUES (in_id_giudice,
            in_id_team, 6);
    END IF;
END;
$$;
```

4.5.12 scoreboard

La seguente funzione consente di visualizzare la classifica finale di un singolo hackathon, mostrando i vincitori e tutti i relativi voti finali.

```
CREATE OR REPLACE FUNCTION scoreboard (
    IN in_hack Hackathon.id_hackathon%TYPE)
RETURNS refcursor
LANGUAGE plpgsql
AS $$
DECLARE
    winners REFCURSOR;
BEGIN
    -- Estraggo dalla classifica globale i team dell'hackathon di riferimento
    OPEN winners for (SELECT nome_team, voto_finale
                      FROM overall_ranking
                      WHERE id_hackathon = in_hack);
    RETURN winners;
END;
$$;
```

4.5.13 invite_judge

La seguente procedura interna si occupa di invitare singolarmente un giudice alla competizione specificata.

```
CREATE OR REPLACE PROCEDURE invite_judge(
    IN in_username Utente.username%TYPE,
    IN in_hack     Hackathon.id_hackathon%TYPE,
    IN in_org      Organizzatore.id_organizzatore%TYPE)
LANGUAGE plpgsql
AS $$
DECLARE
    v_judge Giudice.id_giudice%TYPE;
BEGIN
    SELECT id_giudice INTO v_judge
    FROM Giudice
    WHERE username = in_username;
    IF NOT FOUND THEN -- Verifico l'esistenza del giudice
        INSERT INTO Giudice (username) VALUES (in_username) RETURNING id_giudice
            INTO v_judge;
    END IF;
    INSERT INTO Selezione VALUES (in_hack, in_org, v_judge);
END;
$$;
```

4.5.14 publish_progress

Questa procedura permette ai team di pubblicare un documento sulla piattaforma per aggiornare i giudici dei loro progressi.
Sfrutta il trigger *check_unique_document* per verificare la validità del documento appena creato.

```
CREATE OR REPLACE PROCEDURE publish_progress (
    IN in_team    Team.id_team%TYPE,
    IN in_content Documento.contenuto%TYPE,
    IN in_title   Documento.titolo%TYPE)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Inserisco il documento con i relativi dati
    INSERT INTO Documento (contenuto, titolo, id_team) VALUES (in_content,
        in_title, in_team);
END;
$$;
```

4.5.15 add_hackathon

Procedura per la creazione di un nuovo hackathon all'interno del sistema. Una volta individuati i parametri della competizione, la procedura assegna l'organizzatore corretto (o ne crea uno se necessario) e registra la selezione dei giudici nella base di dati.

```
CREATE OR REPLACE PROCEDURE add_hackathon (
    IN v_title          Hackathon.titolo%TYPE,
    IN v_location       Hackathon.sede%TYPE,
    IN v_start_d        Hackathon.data_inizio%TYPE,
    IN v_end_d          Hackathon.data_fine%TYPE,
    IN v_start_sub      Hackathon.data_apertura_iscrizioni%TYPE,
    IN v_end_sub        Hackathon.data_chiusura_iscrizioni%TYPE,
    IN v_max_sub        Hackathon.max_iscritti%TYPE,
    IN v_max_team_size  Hackathon.max_dim_team%TYPE,
    IN v_username       Utente.username%TYPE,
    IN v_judges         TEXT)
        -- ^ Testo formattato come 'mario.rossi,luca.bianchi,'
LANGUAGE plpgsql
AS $$
DECLARE
    duration  INT;
    v_org     Organizzatore.id_organizzatore%TYPE;
    v_hack    Hackathon.id_hackathon%TYPE;
    pos       INT;
    one_judge Utente.username%TYPE;
BEGIN
    -- Calcolo la durata
    duration := v_end_d - v_start_d;

    -- Verifico l'esistenza dell'organizzatore
    SELECT id_organizzatore INTO v_org
    FROM Organizzatore
    WHERE username = v_username;
    IF NOT FOUND THEN
        INSERT INTO organizzatore (username) VALUES (v_username) RETURNING
            id_organizzatore INTO v_org;
    END IF;

    -- Creo l'hackathon
    INSERT INTO Hackathon (
        titolo, sede, durata, data_inizio, data_fine,
        data_apertura_iscrizioni, data_chiusura_iscrizioni,
        max_iscritti, max_dim_team, id_organizzatore)
        VALUES (v_title, v_location, duration, v_start_d, v_end_d,
            v_start_sub, v_end_sub, v_max_sub, v_max_team_size, v_org)
        RETURNING id_hackathon INTO v_hack;

    -- Inserisco le relative selezioni dei giudici
    LOOP
        pos := INSTR(v_judges, ',');
        EXIT WHEN pos <= 0;
        one_judge := SUBSTR(v_judges, 1, pos-1);
        v_judges := LTRIM(v_judges, one_judge);
        v_judges := LTRIM(v_judges, ',');
        CALL invite_judge(one_judge, v_hack, v_org);
    END LOOP;

EXCEPTION
    WHEN unique_violation THEN
        RAISE EXCEPTION 'In questa sede si e'' gia'' svolto un hackathon con
            questo nome';
END;
$$;
```

4.5.16 delete_user

Procedura per l'eliminazione di un utente dalla piattaforma.

```
CREATE OR REPLACE PROCEDURE delete_user (
    IN username_to_del Utente.username%TYPE)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Verifica se l'utente esiste
    IF NOT EXISTS (SELECT 1
                   FROM Utente
                   WHERE username = username_to_del)
        THEN
        RAISE EXCEPTION 'Utente "%" non trovato', username_to_del;
    END IF;
    -- Elimina solo l'utente (eventuali vincoli ON DELETE CASCADE gestiranno le
        dipendenze se definiti sul DB)
    DELETE FROM Utente WHERE username = username_to_del;
END;
$$;
```

4.5.17 popola_database

La procedura `popola_database()` è stata progettata per inserire dati di esempio coerenti in tutte le tabelle del database, rispettando l'ordine gerarchico delle dipendenze e tutti i vincoli definiti.

4.6 Viste

Introduzione

Un ulteriore strumento utile per un utilizzo efficiente del DB, offerto dai DBMS, è quello delle **viste**:

> *Definizione*: Una vista in un sistema di gestione di database relazionale è una tabella virtuale derivata dal risultato di una query SQL che opera su una o più tabelle base del database.

Il compito delle viste è quello di fornire una visione alternativa degli schemi fisici presenti nel sistema, consentendo di nascondere dati per questioni di riservatezza, creare tabelle NON fisiche con informazioni rilevanti operativamente per il DB e molto altro. Di seguito le viste introdotte nel sistema:

Implementazione delle Viste

4.6.1 overall_ranking

Una vista per visualizzare la classifica globale di tutti i team che hanno partecipato ad una competizione.

```
CREATE OR REPLACE VIEW overall_ranking AS
    SELECT nome AS nome_team, ROUND(AVG(voto), 2) AS voto_finale, id_hackathon
    FROM Votazione NATURAL JOIN Team
    GROUP BY nome_team, id_hackathon
    ORDER BY voto_finale DESC;
```

5 Repository GitHub

Il presente documento è disponibile pubblicamente su GitHub al seguente link:

`github.com/TrialShock26/Hackathon`

La repository contiene:

- Il file per la creazione completa del database
- Questa documentazione di progetto

Fine del Documento

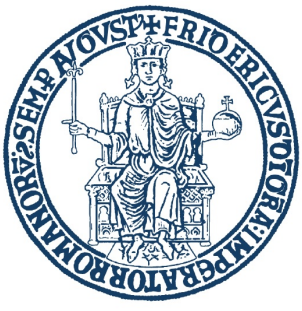
Documentazione Hackathon

Basi Di Dati I

Si ringrazia il lettore per l'attenzione

Mario Majorano	N86005035
Luca Sanselmo	N86005147

Anno Accademico 2024/2025



Università degli Studi
di Napoli Federico II