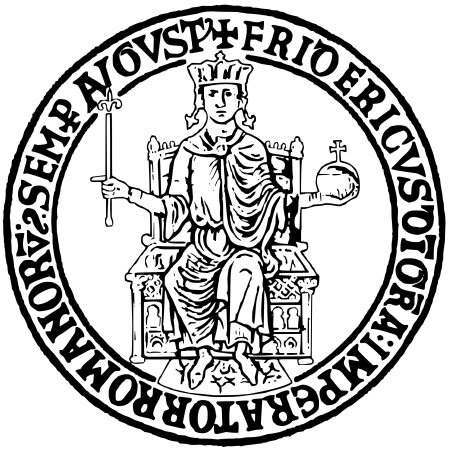
Elaborato di Basi di Dati

"Artemis Cartesio"



Professore: Vincenzo Moscato

Autori: Alessandro Cioffi N46006940
Antonio Cirino N46006930

a.a. 2024/2025

Indice

1 Introduzione

1.1 ArtemisCartesio

ArtemisCartesio rappresenta un'iniziativa pionieristica nel campo dell'esplorazione spaziale, sviluppata da un'agenzia spaziale internazionale per supportare missioni lunari avanzate. L'obiettivo principale è sfruttare tecnologie all'avanguardia per raccogliere, analizzare e gestire i dati provenienti da sensori, robot autonomi e membri dell'equipaggio impegnati in operazioni sulla superficie lunare. Questa piattaforma è progettata per affrontare le complessità logistiche e tecniche di missioni scientifiche, garantendo al contempo efficienza, sicurezza e affidabilità.

Il nome **Artemis Cartesio** unisce il simbolismo della dea Artemide, legata alla luna e all'esplorazione, con il razionalismo scientifico di Cartesio, rappresentando così l'equilibrio tra avventura e metodo. Inoltre, le iniziali **A.C.** omaggiano i creatori del progetto, conferendo un tocco personale al nome.

1.2 Specifiche del Sistema

Il progetto prevede lo sviluppo di una piattaforma informatica che integri:

- **Gestione delle missioni**: Archiviazione di dettagli quali obiettivo, stato, data di inizio e fine, e gestione delle risorse coinvolte.
- **Monitoraggio dei sensori**: Registrazione delle informazioni relative ai sensori (coordinate, tipo, stato operativo) e delle rilevazioni effettuate.
- **Gestione delle anomalie**: Identificazione e registrazione di anomalie, con dati relativi a data, ora, livello di priorità e causa.
- **Interventi e manutenzione**: Programmazione e tracciamento degli interventi per risolvere anomalie, con dettagli quali esito e descrizione delle operazioni effettuate.
- **Reportistica e analisi statistiche**: Compilazione di report relativi alle missioni e analisi statistiche per ottimizzare le operazioni.

1.3 Struttura del Progetto

Per implementare il sistema, il progetto include le seguenti fasi principali:

- **Progettazione della base di dati**:

  - *Concettuale*: Definizione delle entità, delle relazioni e degli attributi principali.
  - *Logica*: Traduzione dello schema concettuale in un modello relazionale.
  - *Fisica*: Ottimizzazione dello schema logico per l'implementazione in Oracle DBMS.

- **Ottimizzazione delle prestazioni**:
  - Creazione di indici per velocizzare le operazioni.
  - Progettazione di strategie di backup, recovery e replicazione per garantire l'affidabilità.

- **Implementazione SQL**:
  - Creazione di stored procedure, trigger, query e viste per gestire le operazioni e supportare l'automazione.

- **Interfaccia web-based**:
  - Sviluppo di un'interfaccia utente tramite Oracle APEX per semplificare l'interazione con il sistema.

1.4 Obiettivi

Il sistema è progettato per:

- Migliorare la gestione e il coordinamento delle missioni lunari.
- Consentire un monitoraggio avanzato e in tempo reale delle operazioni.
- Supportare l'analisi statistica dei dati raccolti per prendere decisioni informate.

Conclusione

Questo progetto non solo rappresenta un passo avanti nella gestione delle missioni spaziali, ma si pone come esempio di integrazione tra tecnologie avanzate e necessità operative. L'approccio metodologico adottato garantisce una base solida per affrontare sfide future nell'esplorazione lunare e spaziale.

2 Creazione DB

In questa sezione si illustra il processo di progettazione e sviluppo del database realizzato dall'Agenzia Spaziale Internazionale ArtemisCartesio. L'architettura del database è stata concepita con l'obiettivo di soddisfare i requisiti operativi del sistema di monitoraggio e controllo delle missioni spaziali, garantendo al contempo una gestione efficiente e una memorizzazione accurata dei dati. Tale sistema comprende informazioni relative alle missioni, ai membri dell'equipaggio, ai sensori, ai robot, alle anomalie, alle rilevazioni e ai report, assicurando una struttura ottimizzata per la loro archiviazione e consultazione.

2.1 Progettazione Concettuale

La **progettazione concettuale** rappresenta la fase iniziale e fondamentale nel processo di progettazione di un database. Durante questa fase, vengono definiti i requisiti del sistema, identificando le entità coinvolte, le relazioni che intercorrono tra di esse e gli attributi che le descrivono. Questo approccio si caratterizza per la sua indipendenza dal modello logico e si focalizza esclusivamente sulla rappresentazione astratta dei dati, senza considerare i dettagli implementativi o tecnologici. Nel contesto di questo progetto, la progettazione concettuale è stata realizzata adottando il **modello Entità/Relazione (E/R)**. Tale modello, grazie alla sua struttura chiara e intuitiva, ha consentito di rappresentare in maniera efficace le entità del sistema, le loro relazioni e i rispettivi attributi, costituendo così una solida base per le fasi successive di progettazione e implementazione del database.

2.1.1 Modello E/R portante

Come prima fase della progettazione concettuale, è stato realizzato uno schema E/R portante. Lo schema portante rappresenta il **nucleo centrale del modello Entità-Relazione (E/R)** per il sistema in esame. Esso evidenzia le principali entità coinvolte: **RISORSA**, **MISSIONE** e **MEMBRO EQUIPAGGIO**, connesse tra loro tramite relazioni chiave.

- La **RISORSA** rappresenta gli strumenti, le tecnologie o gli elementi utilizzati nelle missioni spaziali, essenziali per il loro svolgimento.
- La **MISSIONE** costituisce il fulcro operativo del sistema, in cui le risorse vengono assegnate e gestite, e i membri dell'equipaggio collaborano per il raggiungimento degli obiettivi prefissati
- **MEMBRO EQUIPAGGIO** indica le persone coinvolte nelle missioni, caratterizzate da ruoli specifici e responsabilità ben definite.

Le relazioni delineano i collegamenti logici tra le entità principali, evidenziando i meccanismi di assegnazione e utilizzo delle risorse e del personale nel contesto delle missioni. Questi collegamenti definiscono in modo chiaro e strutturato come le entità interagiscono tra loro, garantendo la coerenza e l'integrità dei dati all'interno del sistema.
Lo schema E/R così elaborato rappresenta una base fondamentale per le successive fasi di progettazione, sia logica che fisica, fornendo una struttura solida e ben definita per l'implementazione del database.

Figure 1: Schema portante del modello E/R

2.1.2 Modello E/R completo

A partire dallo schema portante precedentemente descritto, è stato sviluppato uno schema Entità-Relazione (E/R) completo, che integra tutte le entità e le relazioni coinvolte nel sistema. Questo schema fornisce una rappresentazione esaustiva della struttura del sistema, includendo non solo le principali entità e relazioni, ma anche gli attributi che ne caratterizzano il comportamento e le proprietà.

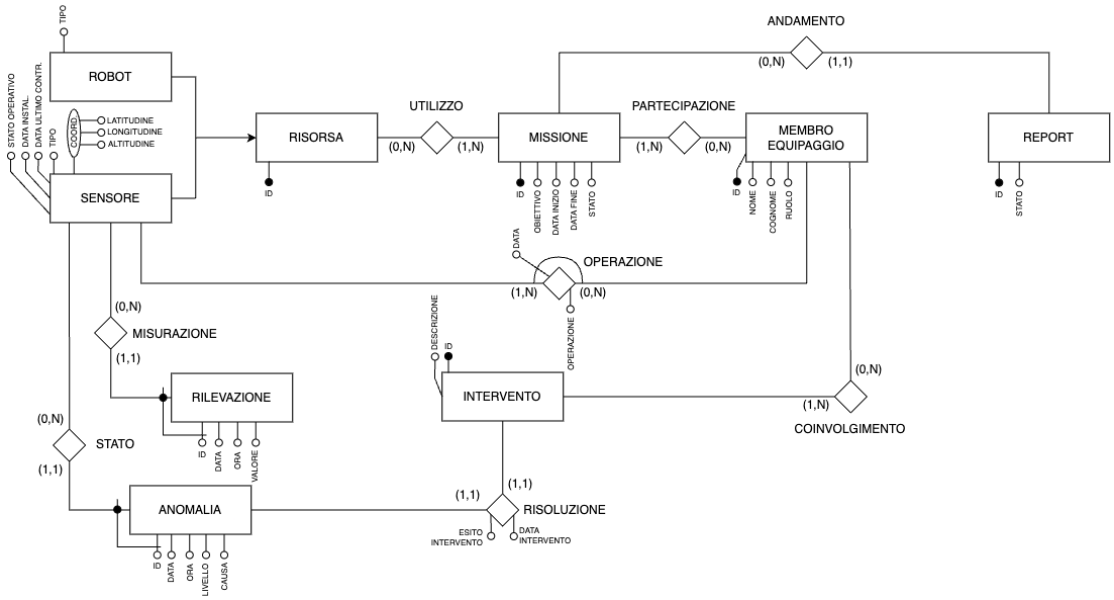


Figure 2: Modello E/R completo

Specifiche di Progettazione

- **MISSIONE**: L'entità **MISSIONE** è identificata da un *ID* e caratterizzata dagli attributi: *Obiettivo*, *Data di inizio*, *Data di fine* e *Stato*. Ogni **MISSIONE** è in relazione con i **MEMBRI DELL'EQUIPAGGIO** tramite l'associazione **PARTECIPAZIONE** con cardinalità $(1, N)$, poiché una missione può coinvolgere uno o più membri dell'equipaggio, mentre un membro può partecipare a zero o più missioni $(0, N)$. L'associazione è di tipo molti-a-molti. Ogni **MISSIONE** è anche in una relazione di **UTILIZZO** con **RISORSA**. Una **MISSIONE** può utilizzare $(1, N)$ risorse (non $(0, N)$ in quanto una missione non avrebbe senso di esistere se non vi fosse associata almeno una risorsa) e una **RISORSA** può essere usata da $(0, N)$ missioni.

- **MEMBRO DELL'EQUIPAGGIO**: L'entità **MEMBRO DELL'EQUIPAGGIO** è identificata da un *Codice univoco* e caratterizzata dagli attributi: *Nome*, *Cognome* e *Ruolo*. Ogni membro dell'equipaggio è responsabile della manutenzione e riparazione dei sensori, partecipando all'associazione **OPERAZIONE** con i **SENSORI** secondo cardinalità $(0, N)$ da parte dell'equipaggio e $(1, N)$ da parte dei sensori. Il tipo di operazione è specificato tramite l'attributo *Operazione* della relazione **OPERAZIONE**. L'attributo *Data* fa sì che un **MEMBRO DELL'EQUIPAGGIO** possa effettuare più operazioni dello stesso tipo sullo stesso sensore nel tempo. Un **MEMBRO DELL'EQUIPAGGIO** può inoltre essere coinvolto tramite la relazione **COINVOLGIMENTO** in $(0, N)$ **INTERVENTO**. Ogni **INTERVENTO** può coinvolgere $(1, N)$ membri.

- **INTERVENTO**: Ogni **INTERVENTO** è avviato per risolvere una specifica **ANOMALIA**. Gli attributi chiave di **INTERVENTO** includono un *ID*, una *Descrizione*, l'*Esito dell'intervento* e la *Data dell'intervento*. La relazione tra **ANOMALIA** e **INTERVENTO**, denominata **RISOLUZIONE**, ha cardinalità $(1, 1)$ su entrambi i lati, garantendo che ogni **INTERVENTO** risolva un'unica **ANOMALIA** e che ogni **ANOMALIA** sia risolta da un unico **INTERVENTO**.

- **REPORT**: Ad ogni **MISSIONE** possono essere associati $(0, N)$ **REPORT** tramite la relazione **ANDAMENTO**. Un **REPORT** è relativo a un'unica **MISSIONE** ed ha dunque cardinalità $(1, 1)$. Gli attributi di **REPORT** includono un *ID* ed uno *Stato*.

- **RILEVAZIONE**: Ogni **RILEVAZIONE** è associata esattamente a un $(1, 1)$ **SENSORE** tramite la relazione denominata **MISURAZIONE**. Un **SENSORE**, tuttavia, può effettuare più rilevazioni, con una cardinalità pari

a $(0, N)$. L'entità **RILEVAZIONE** include una chiave esterna che la collega all'entità **SENSORE** tramite la relazione **MISURAZIONE**, basandosi sugli attributi *Data* e *Ora*. Questi attributi, combinati con l'*ID* del sensore, garantiscono l'identificazione univoca di ciascuna **RILEVAZIONE** effettuata da un determinato **SENSORE**. Si assume che un **SENSORE** non possa eseguire più rilevazioni nello stesso istante temporale definito dalla coppia (*Data*, *Ora*).

- **ANOMALIA**: Ogni **ANOMALIA** è associata esattamente a $(1, 1)$ **SENSORE** tramite la relazione denominata **STATO**. Un **SENSORE**, tuttavia, può presentare più anomalie, con una cardinalità pari a $(0, N)$. L'entità **ANOMALIA** include una chiave esterna che la collega all'entità **SENSORE** tramite la relazione **STATO**, basandosi sugli attributi *Data* e *Ora*. Questi attributi, combinati con l'*ID* del sensore, garantiscono l'identificazione univoca di ciascuna **ANOMALIA** verificatasi in un determinato **SENSORE**. Si assume che in un **SENSORE** non possano verificarsi più anomalie nello stesso istante temporale definito dalla coppia (*Data*, *Ora*).

2.2 Progettazione Logica

La progettazione logica si articola in due fasi:

1. **Trasformazione**: in questa fase, vengono rimossi tutti i costrutti del modello Entità/Relazione (E/R) che non sono direttamente traducibili nel modello logico, come gli attributi composti e gli attributi multi-valore. Gli attributi multi-valore vengono associati direttamente all'entità di partenza, mentre gli attributi composti vengono scomposti nei loro componenti e, se necessario, trasferiti a una nuova entità collegata all'entità originale.

2. **Traduzione**: lo schema risultante dalla trasformazione viene convertito nel modello logico attraverso un insieme di regole predeterminate, che possono essere implementate anche tramite strumenti automatizzati. Questa fase non considera direttamente la semantica dei dati, ma si concentra sulla loro struttura.

2.2.1 Trasformazione

Durante la fase di trasformazione, vengono eliminati tutti gli attributi che non sono direttamente traducibili nel modello logico. Di seguito vengono descritti i casi specifici presenti nello schema:

- **Attributi multi-valore**: non sono presenti in questo caso, quindi non si rende necessaria alcuna operazione di trasformazione relativa a questa tipologia di attributi.
- **Attributi composti**: l'unico attributo composto identificato è **Coordinate**, associato all'entità *SENSORE*. Per conformarsi ai requisiti del modello logico, questo attributo è stato scomposto nei suoi componenti: **Latitudine**, **Longitudine** e **Altitudine**. Tali componenti sono stati direttamente associati all'entità di partenza senza creare una nuova entità.

Di seguito è riportato lo schema trasformato per l'entità *SENSORE*:

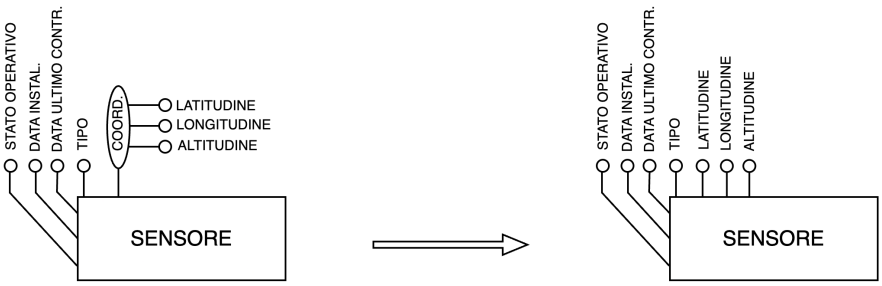


Figure 3: Entità *SENSORE* trasformata

2.2.2 Traduzione

La fase di traduzione consiste nella conversione dello schema Entità/Relazione (E/R), risultato della trasformazione, in uno schema relazionale conforme al modello logico relazionale. Questa operazione garantisce la corretta implementazione delle entità, delle relazioni e degli attributi all'interno del database relazionale.

Traduzione Entità Ogni entità del modello E/R diventa una relazione/tabella.

- **Nome della tabella**: corrisponde al nome dell'entità al plurale.
- **Campi della tabella**: corrispondono agli attributi dell'entità.

Risultato della Traduzione delle Entità:

```
MISSIONI(ID, Obiettivo, Data_Inizio, Data_Fine, Stato);
MEMBRI(ID, Nome, Cognome, Ruolo);
REPORT(ID, Stato);
INTERVENTI(ID, Descrizione);
ANOMALIE(ID, Data, Ora, Livello, Causa);
```

```
RILEVAZIONI(ID, Data, Ora, Valore);
ROBOT(ID, Tipo);
SENSORI(ID, Data_Installazione, Data_Ultimo_Controllo, Tipo,
    Stato_Operativo, Latitudine, Longitudine, Altitudine)
```

Traduzione Relazioni La traduzione delle relazioni tra le entità può essere effettuata attraverso diverse modalità, a seconda delle specifiche esigenze del sistema e delle caratteristiche del modello utilizzato. Di seguito si illustrano le modalità adottate per la progettazione del sistema in esame, con l'obiettivo di garantire una rappresentazione accurata e coerente delle interazioni tra le entità.

1. Relazioni **N a N**

   - Ogni associazione N a N diventa una tabella con:
     - **Nome**: corrisponde al nome dell'associazione, al plurale.
     - **Campi**: includono gli identificatori delle due entità che collega, più eventuali attributi dell'associazione.
     - **Chiave primaria**: composta dalla coppia dei due identificatori.
     - **Vincoli di integrità referenziale**: garantiscono la consistenza con le entità collegate.

```
UTILIZZO_SENSORI (Sensore: Sensori, Missione: Missioni);
UTILIZZO_ROBOT (Robot: Robot, Missione: Missioni);
PARTECIPAZIONI (Missione: Missioni, Membro_Equipaggio:
    Membri_Equipaggio);
OPERAZIONI (Membro_Equipaggio: Membri_Equipaggio, Sensore:
    Sensori, Operazione, Data);
COINVOLGIMENTI (Membro_Equipaggio: Membri_Equipaggio,
    Intervento: Interventi)
```

2. Relazioni **1 a N**

   - L'associazione scompare, **gli attributi della relazione** e **l'identificatore del lato molti** vanno nell'entità lato 1.
   - **Chiave primaria**: Nell'entità lato N rimane quella dell'entità lato N. Nell'entità lato 1 rimane quella del lato 1.
   - Questa scelta consente di ridurre il numero di tabelle, evitando join complessi a 3 tabelle.

```
REPORT (ID, Stato, Data, Missione: Missioni);
RILEVAZIONI (ID, Data, Ora, Valore, Sensore: Sensori);
ANOMALIE (ID, Data, Ora, Livello, Causa, Sensore: Sensori);
```

3. Relazioni **1 a 1**

   - Ogni associazione 1 a 1 diventa una tabella con:
     - **Campi**: includono gli identificatori delle entità che collega, più eventuali attributi.
     - **Chiave primaria**: si sceglie l'identificatore dell'entità con cardinalità minima e partecipazione obbligatoria, per evitare valori NULL. L'altro identificatore deve soddisfare il vincolo di unicità.

```
RISOLUZIONI (Intervento*: Interventi, Anomalia_ID: Anomalie,
   Anomalia_Sensore : Anomalie , Esito_Intervento,
   Data_Intervento)
```

2.2.3 Modello E/R avanzato

Le gerarchie di generalizzazione/specializzazione non possono essere direttamente rappresentate nel modello logico relazionale, poiché quest'ultimo non prevede un costrutto equivalente. Per superare questa limitazione, il modello E/R è stato esteso con l'introduzione del costrutto di **generalizzazione/specializzazione**.

Il costrutto di generalizzazione può essere trasformato in schemi traducibili nel modello logico seguendo tre modalità principali.

Per questo progetto è stata adottata la modalità di **accorpamento della superclasse nelle sottoclassi**:

- **Eliminazione dell'entità padre**: l'entità padre (superclasse) viene eliminata dallo schema.
- **Eredità degli attributi e delle relazioni**: grazie alla proprietà dell'eredità, gli attributi, l'identificatore e le relazioni a cui partecipava l'entità padre vengono trasferiti integralmente alle entità figlie (sottoclassi).

In particolare la specializzazione è di tipo totale-disgiunta:

- **Totale**: Ogni istanza della superclasse deve appartenere ad almeno una sottoclasse. Ogni risorsa deve essere necessariamente o un "sensore" o "robot".
- **Disgiunta**: Un'istanza della superclasse può appartenere a una sola sottoclasse alla volta. Una risorsa può essere un "sensore" o "robot", ma non entrambi contemporaneamente.

In figura (Figura 4) è riportato il risultato dell'accorpamento relativo alla superclasse *RISORSA* e alle sottoclassi *ROBOT* e *SENSORE*.

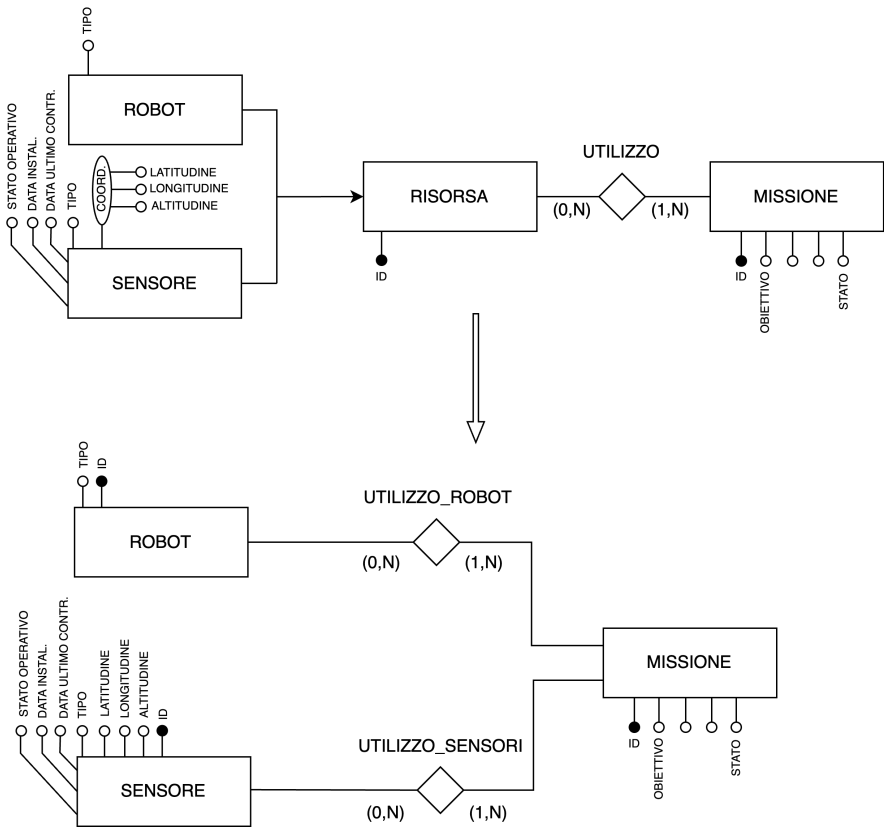


Figure 4: accorpamento di *RISORSA* in *ROBOT* e *SENSORE*

2.3 Progettazione Fisica

I tipi di dato utilizzati per la realizzazione di questo progetto sono:

- **DATE**: Utilizzato per tutte le date presenti;
- **INTEGER**: per tutti gli ID, formati esclusivamente da numeri e non da lettere;
- **VARCHAR2(N)**: per tutti gli attributi di tipo testuali, ad esempio nome, cognome, descrizione, ecc.;
- **FLOAT**: Utilizzato per rappresentare valori numerici con la necessità di precisione decimale, ad esempio coordinate geografiche (latitudine, longitudine, altitudine) o valori rilevati dai sensori;
- **TIMESTAMP**: Utilizzato per rappresentare data e ora in modo dettagliato, includendo secondi e frazioni di secondo, ad esempio per registrare eventi come rilevazioni o anomalie con un'accurata marcatura temporale.

2.3.1 Occupazione tabelle

Viene ora presentata una stima dell'occupazione in byte per le tabelle di un database progettato per un DBMS Oracle. Ogni tabella è analizzata in base ai tipi di dati delle sue colonne.

Tipi di Dati e Dimensioni

- `INT`: 4 byte
- `VARCHAR2(n)`: fino a n byte
- `DATE`: 7 byte
- `TIMESTAMP`: 11 byte
- `FLOAT`: 8 byte

Le dimensioni calcolate sono espresse in byte per riga e in MB per il numero stimato di righe.

- **MISSIONI**: $4 + 255 + 7 + 7 + 50 = 323$ byte, 1.000 righe $\approx 0,31$ MB
- **MEMBRI**: $4 + 100 + 100 + 100 = 304$ byte, 500 righe $\approx 0,14$ MB
- **SENSORI**: $4+7+7+100+50+8+8+8 = 192$ byte, 300 righe $\approx 0,05$ MB
- **ROBOT**: $4 + 100 = 104$ byte, 100 righe $\approx 0,01$ MB
- **ANOMALIE**: $4+7+11+50+255+4 = 331$ byte, 2.000 righe $\approx 0,63$ MB
- **INTERVENTI**: $4 + 255 = 259$ byte, 1.000 righe $\approx 0,25$ MB
- **RISOLUZIONI**: $4 + 4 + 255 + 7 = 270$ byte, 1.000 righe $\approx 0,26$ MB
- **RILEVAZIONI**: $4 + 7 + 11 + 8 + 4 = 34$ byte, 10.000 righe $\approx 0,32$ MB
- **REPORT**: $4 + 50 + 7 + 4 = 65$ byte, 500 righe $\approx 0,03$ MB
- **UTILIZZO_ROBOT**: $4 + 4 = 8$ byte, 1.000 righe $\approx 0,01$ MB
- **UTILIZZO_SENSORI**: $4 + 4 = 8$ byte, 1.000 righe $\approx 0,01$ MB
- **COINVOLGIMENTI**: $4 + 4 = 8$ byte, 500 righe $\approx 0,004$ MB
- **OPERAZIONI**: $4 + 4 + 255 + 7 = 270$ byte, 2.000 righe $\approx 0,51$ MB

- **PARTECIPAZIONI**: $4 + 4 = 8$ byte, 1.000 righe $\approx 0,01$ MB

Le dimensioni teoriche sono calcolate considerando i tipi di dati e ignorando eventuali overhead di gestione. Per una stima complessiva, è necessario moltiplicare queste dimensioni per il numero di righe previste in ciascuna tabella, includendo un margine del 10-20% per l'indicizzazione e altri metadati.

2.3.2 Tabelle

Di seguito viene presentato il codice SQL necessario per la creazione delle tabelle del database, come descritto nelle fasi precedenti.

```
CREATE TABLE MISSIONI (
    ID INT,
    Obiettivo VARCHAR2(255) NOT NULL,
    Data_Inizio DATE NOT NULL,
    Data_Fine DATE,
    Stato VARCHAR2(50) NOT NULL
);

CREATE TABLE MEMBRI (
    ID INT,
    Nome VARCHAR2(100) NOT NULL,
    Cognome VARCHAR2(100) NOT NULL,
    Ruolo VARCHAR2(100) NOT NULL
);

CREATE TABLE SENSORI (
    ID INT,
    Data_Installazione DATE NOT NULL,
    Data_Ultimo_Controllo DATE,
    Tipo VARCHAR2(100) NOT NULL,
    Stato_Operativo VARCHAR2(50) NOT NULL,
    Latitudine FLOAT NOT NULL,
    Longitudine FLOAT NOT NULL,
    Altitudine FLOAT NOT NULL
);

CREATE TABLE ROBOT (
    ID INT,
    Tipo VARCHAR2(100) NOT NULL
);

CREATE TABLE ANOMALIE (
    ID INT,
    Data DATE NOT NULL,
    Ora TIMESTAMP NOT NULL,
    Livello VARCHAR2(50) NOT NULL,
```

```
    Causa VARCHAR2(255) NOT NULL,
    Sensore INT
);

CREATE TABLE INTERVENTI (
    ID INT,
    Descrizione VARCHAR2(255) NOT NULL
);

CREATE TABLE RISOLUZIONI (
    Anomalia_ID INT,
    Anomalia_Sensore INT,
    Intervento INT,
    Esito_Intervento VARCHAR2(255),
    Data_Intervento DATE NOT NULL,
    UNIQUE(Intervento)
);

CREATE TABLE RILEVAZIONI (
    ID INT,
    Data DATE NOT NULL,
    Ora TIMESTAMP NOT NULL,
    Valore FLOAT NOT NULL,
    Sensore INT
);

CREATE TABLE REPORT (
    ID INT,
    Stato VARCHAR2(50) NOT NULL,
    Data DATE,
    Missione INT
);

CREATE TABLE UTILIZZO_ROBOT (
    Robot INT,
    Missione INT
);

CREATE TABLE UTILIZZO_SENSORI (
    Sensore INT,
    Missione INT
);

CREATE TABLE COINVOLGIMENTI (
    Membro INT,
    Intervento INT
);

CREATE TABLE OPERAZIONI (
```

```
    Membro INT,
    Sensore INT,
    Operazione VARCHAR2(255),
    Data DATE NOT NULL
);

CREATE TABLE PARTECIPAZIONI (
    Missione INT,
    Membro INT
);
```

2.3.3 Chiavi primarie

Di seguito viene riportato il codice SQL necessario per la creazione dei vincoli di chiave primaria utilizzando la notazione "`PK_NomeTabella`" per una maggiore chiarezza e standardizzazione e per aiutare l'individuazione di eventuali errori.

```
ALTER TABLE MISSIONI ADD CONSTRAINT PK_MISSIONI PRIMARY KEY (ID);
ALTER TABLE MEMBRI ADD CONSTRAINT PK_MEMBRI PRIMARY KEY (ID);
ALTER TABLE SENSORI ADD CONSTRAINT PK_SENSORI PRIMARY KEY (ID);
ALTER TABLE ROBOT ADD CONSTRAINT PK_ROBOT PRIMARY KEY (ID);
ALTER TABLE ANOMALIE ADD CONSTRAINT PK_ANOMALIE PRIMARY KEY (ID,
    Sensore);
ALTER TABLE INTERVENTI ADD CONSTRAINT PK_INTERVENTI PRIMARY KEY (
    ID);
ALTER TABLE RILEVAZIONI ADD CONSTRAINT PK_RILEVAZIONI PRIMARY KEY
    (ID, Sensore);
ALTER TABLE REPORT ADD CONSTRAINT PK_REPORT PRIMARY KEY (ID);
ALTER TABLE RISOLUZIONI ADD CONSTRAINT PK_RISOLUZIONI PRIMARY KEY
    (Anomalia_ID, Anomalia_Sensore);
ALTER TABLE UTILIZZO_ROBOT ADD CONSTRAINT PK_UTILIZZO_ROBOT
    PRIMARY KEY (Robot, Missione);
ALTER TABLE UTILIZZO_SENSORI ADD CONSTRAINT PK_UTILIZZO_SENSORI
    PRIMARY KEY (Sensore, Missione);
ALTER TABLE COINVOLGIMENTI ADD CONSTRAINT PK_COINVOLGIMENTI
    PRIMARY KEY (Membro, Intervento);
ALTER TABLE OPERAZIONI ADD CONSTRAINT PK_OPERAZIONI PRIMARY KEY (
    Membro, Sensore, Data);
ALTER TABLE PARTECIPAZIONI ADD CONSTRAINT PK_PARTECIPAZIONI
    PRIMARY KEY (Missione, Membro);
```

2.3.4 Chiavi esterne

Di seguito viene riportato il codice SQL necessario per la creazione dei vincoli di chiave esterna utilizzando la notazione "`FK_TabellaReferenziante_TabellaReferenziata`".

```
1 ALTER TABLE ANOMALIE ADD CONSTRAINT FK_ANOMALIE_SENSORI FOREIGN
    KEY (Sensore) REFERENCES SENSORI(ID) ON DELETE CASCADE;
2 ALTER TABLE RISOLUZIONI ADD CONSTRAINT FK_RISOLUZIONI_ANOMALIE
    FOREIGN KEY (Anomalia_ID, Anomalia_Sensore ) REFERENCES
    ANOMALIE(ID, Sensore) ON DELETE CASCADE;
3 ALTER TABLE RISOLUZIONI ADD CONSTRAINT FK_RISOLUZIONI_INTERVENTI
    FOREIGN KEY (Intervento) REFERENCES INTERVENTI(ID) ON DELETE
    CASCADE;
4 ALTER TABLE REPORT ADD CONSTRAINT FK_REPORT_MISSIONI FOREIGN KEY (
    Missione) REFERENCES MISSIONI(ID)ON DELETE CASCADE;
5 ALTER TABLE UTILIZZO_ROBOT ADD CONSTRAINT FK_UTILIZZO_ROBOT_ROBOT
    FOREIGN KEY (Robot) REFERENCES ROBOT(ID) ON DELETE CASCADE;
6 ALTER TABLE UTILIZZO_ROBOT ADD CONSTRAINT
    FK_UTILIZZO_ROBOT_MISSIONI FOREIGN KEY (Missione) REFERENCES
    MISSIONI(ID)ON DELETE CASCADE;
7 ALTER TABLE UTILIZZO_SENSORI ADD CONSTRAINT
    FK_UTILIZZO_SENSORI_SENSORI FOREIGN KEY (Sensore) REFERENCES
    SENSORI(ID) ON DELETE CASCADE;
8 ALTER TABLE UTILIZZO_SENSORI ADD CONSTRAINT
    FK_UTILIZZO_SENSORI_MISSIONI FOREIGN KEY (Missione) REFERENCES
    MISSIONI(ID)ON DELETE CASCADE;
9 ALTER TABLE COINVOLGIMENTI ADD CONSTRAINT FK_COINVOLGIMENTI_MEMBRI
     FOREIGN KEY (Membro) REFERENCES MEMBRI(ID)ON DELETE CASCADE;
10 ALTER TABLE COINVOLGIMENTI ADD CONSTRAINT
    FK_COINVOLGIMENTI_INTERVENTI FOREIGN KEY (Intervento)
    REFERENCES INTERVENTI(ID)ON DELETE CASCADE;
11 ALTER TABLE OPERAZIONI ADD CONSTRAINT FK_OPERAZIONI_MEMBRI FOREIGN
     KEY (Membro) REFERENCES MEMBRI(ID)ON DELETE CASCADE;
12 ALTER TABLE OPERAZIONI ADD CONSTRAINT FK_OPERAZIONI_SENSORI
    FOREIGN KEY (Sensore) REFERENCES SENSORI(ID)ON DELETE CASCADE;
13 ALTER TABLE PARTECIPAZIONI ADD CONSTRAINT
    FK_PARTECIPAZIONI_MISSIONE FOREIGN KEY (Missione) REFERENCES
    MISSIONI(ID)ON DELETE CASCADE;
14 ALTER TABLE PARTECIPAZIONI ADD CONSTRAINT FK_PARTECIPAZIONI_MEMBRI
     FOREIGN KEY (Membro) REFERENCES MEMBRI(ID)ON DELETE CASCADE;
```

L'opzione `ON DELETE CASCADE` consente di configurare il database affinché, quando un record nella tabella primaria viene eliminato, tutti i record nelle tabelle che lo referenziano tramite chiavi esterne vengano automaticamente rimossi. Questo meccanismo garantisce l'integrità referenziale del database, evitando la presenza di dati orfani che potrebbero comprometterne la coerenza. Inoltre, semplifica la gestione delle relazioni, eliminando la necessità di interventi manuali per la rimozione dei dati correlati. Tuttavia, è necessario adottare questa opzione con cautela, poiché l'eliminazione di un singolo record potrebbe propagarsi su larga scala, con conseguenze potenzialmente indesiderate.

2.3.5 Vincoli di check

In fine, si riportano i comandi SQL per la creazione di vincoli di check utilizzando la notazione "`CK_Tabella_Campo`". Questo approccio consente di garantire che i valori di determinati campi rispettino i criteri specificati, fornendo così un ulteriore livello di controllo sull'integrità dei dati.

```
1  ALTER TABLE SENSORI ADD CONSTRAINT CK_SENSORI_TIPO CHECK (Tipo IN
       ('Temperatura', 'Pressione', 'Gas', 'Radiazioni', 'Geologia'));
2
3  ALTER TABLE SENSORI ADD CONSTRAINT CK_SENSORI_STATO CHECK (
       Stato_Operativo IN ('Attivo', 'Standby', 'Manutenzione', '
       Malfunzionante'));
4
5
6  ALTER TABLE OPERAZIONI ADD CONSTRAINT CK_OPERAZIONI_STATO CHECK (
       Stato_Operativo IN ('Attivo', 'Standby', 'Manutenzione', '
       Malfunzionante'));
7
8  ALTER TABLE ANOMALIE ADD CONSTRAINT CK_ANOMALIE_LIVELLO CHECK (
       Livello IN ('Bassa', 'Media', 'Alta', 'Critica'));
9
10 ALTER TABLE MISSIONI ADD CONSTRAINT CK_MISSIONI_STATO CHECK (Stato
        IN ('Pianificata', 'In corso', 'Completata', 'Annullata'));
```

N.B.: I quattro snippet di codice SQL mostrati verranno implementati nel DBMS tramite appositi script contenenti il codice: (1) `create_table.sql`, (2) `constraint.sql`.

3 Ottimizzazione DB

3.1 Indici

Un **indice** in un sistema di gestione di database (DBMS) è una struttura dati organizzata che consente di individuare rapidamente un determinato record all'interno di un file di dati. Uno dei principali vantaggi dell'utilizzo degli indici è il **miglioramento delle prestazioni delle query**: gli indici permettono di ridurre il tempo necessario per cercare i dati, evitando una scansione completa della tabella.

Creare indici sui campi che vengono frequentemente utilizzati nei filtri delle query (ad esempio, con condizioni `WHERE`, `JOIN`, `ORDER BY` o `GROUP BY`) può velocizzare significativamente l'elaborazione delle richieste, ottimizzando il sistema nel suo complesso.

Tuttavia, è importante bilanciare l'uso degli indici per evitare costi aggiuntivi durante le operazioni di scrittura come `INSERT`, `UPDATE` e `DELETE`, poiché gli indici devono essere aggiornati ogni volta che i dati della tabella vengono modificati.

Tralasciando gli indici sulla chiave primaria, in quanto il DBMS crea un indice per ogni chiave primaria della tabella, abbiamo ritenuto opportuna l'aggiunta dei seguenti indici:

```
-- Per filtrare missioni sullo stato
CREATE INDEX idx_missioni_stato ON MISSIONI(Stato);

-- Per filtrare sul ruolo dei membri
CREATE INDEX idx_membri_ruolo ON MEMBRI(Ruolo);

-- Per filtrare sul tipo di robot
CREATE INDEX idx_robot_tipo ON ROBOT(Tipo);

-- Per filtrare sulla data del report
CREATE INDEX idx_report_data ON REPORT(Data);

-- Per filtrare sulla coppia sensore-data di una rilevazione
CREATE INDEX idx_rilevazioni_sensori_data ON RILEVAZIONI(Sensore,
    Data);
```

3.2 Concorrenza

Per la gestione della concorrenza, si è scelto di adottare il protocollo **2PL stretto** (*Strict Two-Phase Locking*), una variante del protocollo **2PL** (*Two-Phase Locking*). Entrambi i protocolli si basano sul meccanismo dei **lock**, il quale consente di controllare l'accesso concorrente agli oggetti condivisi e garantisce la **serializzabilità delle transazioni**, assicurando che il risultato delle operazioni concorrenti

sia equivalente a una loro esecuzione sequenziale.
Il protocollo 2PL stretto segue le seguenti regole operative:

- `lock()`: ogni oggetto coinvolto nelle operazioni è protetto da un lock per controllare l'accesso concorrente.;
- `read_lock()`: quando una transazione desidera leggere un oggetto, viene applicato un lock di lettura. Questo tipo di lock consente a più transazioni di leggere contemporaneamente lo stesso oggetto, permettendo una condivisione sicura;
- `write_lock()`: Quando una transazione desidera modificare un oggetto, viene applicato un lock di scrittura. Questo lock è esclusivo, consentendo a una sola transazione per volta di effettuare modifiche sull'oggetto;
- `unlock()`: ogni lock deve essere rilasciato una volta terminata l'operazione corrispondente, consentendo ad altre transazioni di accedere all'oggetto;

Gli oggetti possono trovarsi in tre stati: **libero**, **bloccato in lettura**, o **bloccato in scrittura**.

La scelta del **2PL stretto** rispetto al **2PL** è dovuta al fatto che il **2PL stretto** evita l'anomalia delle **letture sporche** (*dirty reads*), che possono verificarsi in altri approcci di gestione della concorrenza.

Il protocollo 2PL stretto (Strict Two-Phase Locking) si articola in due fasi principali, che regolano l'acquisizione e il rilascio dei lock da parte delle transazioni:

1. **Fase crescente**
    - In questa fase, la transazione acquisisce tutte le risorse necessarie mediante i comandi `read_lock` e `write_lock`.
    - La fase crescente termina quando la transazione ha acquisito tutti i lock necessari per completare le proprie operazioni.
2. **Fase decrescente**
    - Questa fase inizia quando la transazione rilascia il primo lock mediante il comando `unlock`.
    - Nel caso del protocollo **2PL stretto**, il rilascio dei lock (fase decrescente) può avvenire esclusivamente dopo che la transazione ha eseguito il **commit** o l'**abort**, garantendo che le risorse modificate non siano accessibili da altre transazioni fino a quando non sono consolidate o annullate.

Questo approccio garantisce che le transazioni siano serializzabili, impedendo conflitti e mantenendo la consistenza dei dati.

3.3 Affidabilità

Il controllo di **affidabilità** in un sistema di basi di dati ha come obiettivo principale il **ripristino** dello stato corretto del sistema (recovery) in seguito a guasti accidentali o intenzionali, che possano compromettere la funzionalità del sistema stesso. I guasti possono essere legati sia a malfunzionamenti hardware (ad esempio, guasti su disco o memoria) che software (ad esempio, crash di applicazioni o errori di sistema).

Il sistema di affidabilità si basa sulla gestione delle **transazioni**, che sono le unità fondamentali delle operazioni nel database, garantendo **atomicità** (le transazioni sono eseguite in modo completo o non eseguite affatto) e **persistenza** (i dati delle transazioni devono essere memorizzati in modo permanente una volta che la transazione è stata completata correttamente).

3.3.1 Backup

Per garantire un alto livello di affidabilità, il nostro sistema di database implementa la strategia di backup RAID 1, che offre una soluzione di mirroring. In un sistema RAID 1, ogni dato scritto sul disco primario viene duplicato in tempo reale su un disco secondario, chiamato "mirror". Questa tecnica garantisce che, in caso di guasto di uno dei dischi, i dati siano ancora disponibili sull'altro disco, riducendo il rischio di perdita di informazioni e migliorando la disponibilità del sistema.

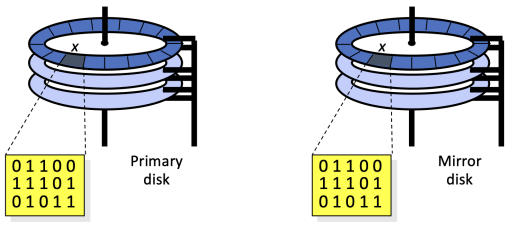


Figure 5: RAID 1 Mirroring

3.3.2 Recovery

Il gestore dell'affidabilità deve gestire l'esecuzione dei comandi transazionali di `begin transaction`, `commit`, `rollback` e tutte le operazioni di ripristino dopo i guasti.

Per poter effettuare ciò, il gestore deve possedere un file di log: un file presente su memoria stabile che registra tutte le operazioni svolte dalle transazioni nel loro ordine di esecuzione.

Il log è quindi una sorta di "diario di bordo" che, in un qualsiasi istante, permette di ricostituire il contenuto corretto della base dei dati a seguito di malfunzionamenti.

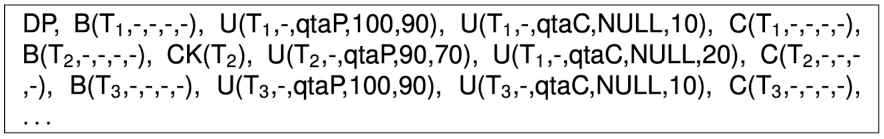


Figure 6: File di log

Tecniche di recovery

- **Ripresa a freddo**: In caso di guasti hardware che interessano i dispositivi di memoria di massa (ad esempio, guasti ai dischi rigidi), si verifica la perdita del contenuto sia della memoria centrale che di quella secondaria. Tuttavia, la memoria stabile, come i dispositivi di backup, rimane intatta. In tali circostanze, si procede con una **ripresa a freddo** (*cold restart*), che comporta un ripristino approfondito. Questo processo richiede il recupero dei dati mediante l'uso di backup e log disponibili.

- **Ripresa a caldo**: Nei casi di guasti software (come errori di programma, crash di sistema, interruzioni dell'alimentazione, ecc.), viene compromesso esclusivamente il contenuto della memoria centrale, mentre la memoria secondaria e quella stabile rimangono intatte. In queste situazioni si procede con la **ripresa a caldo** (*warm restart*), che consente un ripristino più rapido rispetto alla ripresa a freddo.

Indipendentemente dal tipo di ripresa adottata, la procedura di recovery segue tre fasi principali, secondo il **Modello Fail-Stop**:

1. **Arresto delle transazioni attive**: Tutte le transazioni attualmente in esecuzione sul sistema di basi di dati vengono forzatamente interrotte per evitare ulteriori inconsistenze.

2. **Ripristino del sistema operativo**: Si procede al riavvio e alla verifica del corretto funzionamento del sistema operativo, necessario per garantire l'esecuzione della fase successiva.

3. **Esecuzione del ripristino**: Viene avviata la procedura di recupero, che sfrutta backup, log e informazioni residue per ripristinare la consistenza e l'integrità del database.

Queste tecniche, basate sul modello **fail-stop**, mirano a garantire la continuità operativa e la salvaguardia dei dati, anche in presenza di guasti critici.

4 Meccanismi avanzati

I moderni sistemi di gestione di basi di dati (DBMS) offrono una vasta gamma di strumenti e funzionalità avanzate progettate per migliorare l'efficienza, l'automazione e la personalizzazione nella gestione e nell'elaborazione dei dati. Questi meccanismi avanzati rappresentano componenti fondamentali per lo sviluppo di applicazioni scalabili, sicure e performanti, in grado di rispondere alle esigenze sempre più complesse degli ambienti operativi.

In questo capitolo verrò approfondito l'uso, relativo al progetto, di:

- **Query**: tecniche per ottimizzare l'interrogazione e la manipolazione dei dati, sfruttando funzionalità avanzate del linguaggio SQL.
- **View**: viste logiche che permettono di semplificare l'accesso ai dati e migliorare la sicurezza;
- **Stored Procedures**: programmi precompilati memorizzati nel database, utili per eseguire operazioni complesse in modo efficiente;
- **Trigger**: meccanismi che consentono di automatizzare l'esecuzione di azioni in risposta a eventi specifici nel database;

4.1 Query

Query N.1 Anomalie con livello 'CRITICO'

```
SELECT
    A.ID AS ID_SENSORE,
    A.CAUSA AS CAUSA_ANOMALIA,
    S.TIPO AS TIPO_SENSORE
FROM
    ANOMALIE A
JOIN
    SENSORI S ON A.SENSORE = S.ID
WHERE
    A.LIVELLO = 'Critica'
ORDER BY
    A.ID ASC;
```

Query N.2 Numero di robot utilizzati da ogni missione

```
SELECT
    M.ID AS ID_MISSIONE,
    M.OBIETTIVO AS OBIETTIVO_MISSIONE,
    COUNT(*) AS NUMERO_ROBOT_MISSIONE
FROM
```

```
    UTILIZZO_ROBOT UR
JOIN
    ROBOT R ON UR.Robot = R.ID
JOIN
    MISSIONI M ON M.ID = UR.Missione
GROUP BY
    M.ID, M.OBIETTIVO
ORDER BY
    M.ID ASC;
```

ID SENSORE	CAUSA ANOMALIA	TIPO SENSORE
3	Guasto Hardware	Temperatura
6	Problemi di connessione	Gas

Table 1: Esempio di risultati della Query N.1

ID MISSIONE	OBIETTIVO MISSIONE	NUMERO ROBOT
1	Esplorazione Polo Sud Lunare	3
2	Studio Crateri Lunari	4

Table 2: Esempio di risultati della Query N.2

Query N.3 Causa principale di anomalie

```
SELECT
    A.CAUSA,
    COUNT(*) AS NUM_VOLTE
FROM
    ANOMALIE A
GROUP BY
    A.CAUSA
HAVING COUNT(*) = (
    SELECT MAX(NUM_CAUSE)
    FROM (
        SELECT A.CAUSA, COUNT(*) AS NUM_CAUSE
        FROM ANOMALIE A
        GROUP BY A.CAUSA
    )
);
```

Query N.4 Il numero di membri coinvolto in ciascuna operazione, per ogni sensore

```
SELECT
    S.ID AS ID_SENSORE,
    O.OPERAZIONE, O.DATA AS DATA_OPERAZIONE,
    COUNT(O.MEMBRO) AS NUM_MEMBRI
FROM
    OPERAZIONI O
JOIN
    MEMBRI M ON O.MEMBRO = M.ID
JOIN
    SENSORI S ON O.SENSORE = S.ID
GROUP BY
    S.ID, O.OPERAZIONE, O.DATA
ORDER BY
    O.DATA;
```

Query N.5 Il numero di interventi effettuati da ogni membro dell'equipaggio

```
SELECT
    M.ID, CONCAT(M.NOME, CONCAT(' ', M.COGNOME)) AS NOME_COMPLETO,
    COUNT(*) AS NUM_INTERVENTI_EFFETTUATI
FROM
    COINVOLGIMENTI C
JOIN
    MEMBRI M ON C.MEMBRO = M.ID
GROUP BY
    M.ID, M.NOME, M.COGNOME
ORDER BY
    M.ID ASC;
```

ID	NOME COMPLETO	NUM INTERVENTI EFFETTUATI
1	Luca Rossi	1
6	Sara Russo	2

Table 3: Esempio di risultati della Query N.5

Query N.6 Sensori che non hanno mai avuto anomalie

```
SELECT
    ID
FROM
    SENSORI
```

```
WHERE
    ID NOT IN
    (SELECT S.ID
    FROM SENSORI S
    JOIN ANOMALIE A ON S.ID = A.SENSORE
    GROUP BY S.ID);
```

4.2 View

View N.1 Per la visualizzazione dei membri (nome, cognome) e delle missioni a cui partecipano (ID, obiettivo, stato)

```
CREATE VIEW MEMBRI_MISSIONI AS
SELECT
    CONCAT(M.NOME, CONCAT(' ', M.COGNOME)) AS NOME_COMPLETO,
    MI.OBIETTIVO AS OBIETTIVO_MISSIONE,
    MI.ID AS ID_MISSIONE,
    MI.STATO AS STATO_MISSIONE
FROM
    PARTECIPAZIONI P
JOIN
    MEMBRI M ON P.MEMBRO = M.ID
JOIN
    MISSIONI MI ON P.MISSIONE = MI.ID;
```

View N.2 Per la visualizzazione dei robot (tipo) e delle missioni a cui partecipano (obiettivo)

```
CREATE VIEW ROBOT_MISSIONI AS
SELECT
    R.ID AS ID_ROBOT,
    R.TIPO AS TIPO_ROBOT,
    M.ID AS ID_MISSIONE,
    M.OBIETTIVO AS OBIETTIVO_MISSIONE
FROM
    UTILIZZO_ROBOT UR
JOIN
    ROBOT R ON UR.ROBOT = R.ID
JOIN
    MISSIONI M ON UR.MISSIONE = M.ID;
```

View N.3 Per la visualizzazione dei sensori (tipo, stato operativo) e dei membri (nome, cognome) che effettuano operazioni (tipo, data) su di essi

```
CREATE VIEW SENSORI_MISSIONI AS
SELECT
    CONCAT(M.NOME, CONCAT(' ', M.COGNOME)) AS NOME_COMPLETO,
```

```
    S.TIPO AS TIPO_SENSORE,
    S.STATO_OPERATIVO AS STATO_SENSORE,
    O.OPERAZIONE AS TIPO_OPERAZIONE,
    O.DATA
FROM
    OPERAZIONI O
JOIN
    MEMBRI M ON O.MEMBRO = M.ID
JOIN
    SENSORI S ON O.SENSORE = S.ID;
```

View N.4 Per la visualizzazione delle anomalie rilevate per ciascun sensore

```
CREATE VIEW ANOMALIE_SENSORI AS
SELECT
    S.ID AS ID_SENSORE,
    S.TIPO AS TIPO_SENSORE,
    S.STATO_OPERATIVO AS STATO_SENSORE,
    A.ID AS ID_ANOMALIA,
    A.LIVELLO AS LIVELLO_ANOMALIA,
    A.CAUSA AS CAUSA_ANOMALIA,
    A.DATA,
    TO_CHAR(A.ORA, 'HH24:MI:SS') AS ORA
FROM
    ANOMALIE A
JOIN
    SENSORI S ON A.SENSORE = S.ID;
```

View N.5 Per vedere il numero di missione a cui un membro partecipa

```
CREATE VIEW PARTECIPAZIONI_MEMBRI AS
SELECT
    M.ID AS ID_MEMBRO,
    CONCAT(CONCAT(M.NOME, ' '), M.COGNOME) AS NOME_COMPLETO,
    COUNT(*) AS NUMERO_PARTECIPAZIONI
FROM
    PARTECIPAZIONI P
JOIN
    MEMBRI M ON P.MEMBRO = M.ID
JOIN
    MISSIONI MI ON P.MISSIONE = MI.ID
GROUP BY
    M.ID, M.NOME, M.COGNOME
ORDER BY
    M.ID ASC;
```

View N.6 Per vedere i report di ogni missione

```sql
CREATE VIEW ANDAMENTO_MISSIONI AS
SELECT
    M.ID AS ID_MISSIONE,
    M.OBIETTIVO AS OBIETTIVO_MISSIONE,
    M.STATO AS STATO_MISSIONE,
    R.ID AS ID_REPORT,
    R.STATO AS STATO_REPORT,
    R.DATA AS DATA_REPORT
FROM
    REPORT R
    JOIN MISSIONI M ON R.MISSIONE = M.ID
ORDER BY
    M.ID ASC;
```

4.3 Stored Procedure

Le **stored procedures** rappresentano uno degli strumenti più potenti e versatili messi a disposizione dai moderni sistemi di gestione di basi di dati (DBMS). Si tratta di programmi precompilati e memorizzati direttamente nel database, che consentono di eseguire operazioni complesse in modo efficiente e sicuro. Grazie alla loro capacità di combinare logica applicativa e accesso ai dati, le stored procedures sono ampiamente utilizzate in ambiti che richiedono elevati livelli di prestazioni, sicurezza e manutenzione centralizzata del codice.

Di seguito sono riportate le stored procedures adottate per il progetto.

Procedure N.1 Per assegnare un sensore ad una missione, con relativi controlli aggiuntivi.

```sql
CREATE OR REPLACE PROCEDURE AssegnareSensoreAMissione(
    p_Sensore_ID IN NUMBER,
    p_Missione_ID IN NUMBER
) AS
    v_count_sensore NUMBER;
    v_count_missione NUMBER;
BEGIN
    -- Incrementa il valore di v_count_sensore se esso è presente
    nel database
    SELECT COUNT(*)
    INTO v_count_sensore
    FROM SENSORI
    WHERE ID = p_Sensore_ID;

    -- Incrementa il valore di v_count_missione se esso è presente
    nel database
```

```
    SELECT COUNT(*)
    INTO v_count_missione
    FROM MISSIONI
    WHERE ID = p_Missione_ID;

    -- Verifica se essi sono presenti o meno nel databse altrimenti si genera errore
    IF v_count_sensore = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Errore: il sensore con ID ' p_Sensore_ID  ' non esiste.');
    ELSIF v_count_missione = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Errore: la missione con ID ' p_Missione_ID  ' non esiste.');
    ELSE
        -- Verifica se il sensore è già assegnato alla missione
        SELECT COUNT(*)
        INTO v_count_sensore
        FROM UTILIZZO_SENSORI
        WHERE Sensore = p_Sensore_ID AND Missione = p_Missione_ID;

        IF v_count_sensore = 0 THEN
            -- Se il sensore non è già assegnato, inserisci
            INSERT INTO UTILIZZO_SENSORI (Sensore, Missione)
            VALUES (p_Sensore_ID, p_Missione_ID);

            DBMS_OUTPUT.PUT_LINE('Sensore '  p_Sensore_ID  ' assegnato alla missione '  p_Missione_ID);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Il sensore '  p_Sensore_ID  ' è già assegnato alla missione '  p_Missione_ID);
        END IF;
    END IF;
END;
```

```
EXECUTE AssegnareSensoreAMissione(11, 1);
```

```
Sensore 11 assegnato alla missione 1.
```

```
EXECUTE AssegnareSensoreAMissione(30, 5);
```

```
Errore:  il sensore con ID 30 non esiste.
```

Procedure N.2 Per inserire un membro nella tabella coinvolgimenti relativamente ad un intervento

```
CREATE OR REPLACE PROCEDURE InserireMembroInCoinvolgimento(
    p_Membro_ID IN NUMBER,
    p_Intervento_ID IN NUMBER
) AS
    v_count_membro NUMBER;
```

```
    v_count_intervento NUMBER;
    v_count_coinvolgimento NUMBER;
BEGIN
    -- Verifica se il membro esiste nel database
    SELECT COUNT(*)
    INTO v_count_membro
    FROM MEMBRI
    WHERE ID = p_Membro_ID;

    IF v_count_membro = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Errore: il membro con ID '
    p_Membro_ID  ' non esiste nel database.');
    ELSE
        -- Verifica se l'intervento esiste nel database
        SELECT COUNT(*)
        INTO v_count_intervento
        FROM INTERVENTI
        WHERE ID = p_Intervento_ID;

        IF v_count_intervento = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Errore: l''intervento con ID '
    p_Intervento_ID  ' non esiste nel database.');
        ELSE
            -- Verifica se il membro è già coinvolto nell'
    intervento
            SELECT COUNT(*)
            INTO v_count_coinvolgimento
            FROM COINVOLGIMENTI
            WHERE Membro = p_Membro_ID AND Intervento =
    p_Intervento_ID;

            IF v_count_coinvolgimento > 0 THEN
                DBMS_OUTPUT.PUT_LINE('Errore: il membro con ID '
    p_Membro_ID  ' è già coinvolto nell''intervento con ID '
    p_Intervento_ID);
            ELSE
                -- Inserisce il membro nell'intervento se non è gi
    à coinvolto
                INSERT INTO COINVOLGIMENTI (Membro, Intervento)
                VALUES (p_Membro_ID, p_Intervento_ID);

                DBMS_OUTPUT.PUT_LINE('Membro con ID '  p_Membro_ID
      ' aggiunto all''intervento con ID '  p_Intervento_ID);
            END IF;
        END IF;
    END IF;
END;
```

```
EXECUTE InserireMembroInCoinvolgimento(15, 10);
```

Membro con ID 15 aggiunto all'intervento con ID 10

```
EXECUTE InserireMembroInCoinvolgimento(1, 1);
```

Errore: il membro con ID 1 è già coinvolto nell'intervento con ID 1

Procedure N.3 Per aggiornare lo stato operativo di un sensore, con controlli di sicurezza.

```
CREATE OR REPLACE PROCEDURE AggiornareStatoSensore(
    p_Sensore_ID IN NUMBER,
    p_Nuovo_Stato IN VARCHAR2
) AS
    v_count_sensore NUMBER;
    v_stato_valido BOOLEAN := FALSE;
BEGIN
    -- Verifica se il sensore esiste nel database
    SELECT COUNT(*)
    INTO v_count_sensore
    FROM SENSORI
    WHERE ID = p_Sensore_ID;

    IF v_count_sensore = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Errore: il sensore con ID '
    p_Sensore_ID  ' non esiste nel database.');
    ELSE
        -- Controllo se il nuovo stato è valido
        IF p_Nuovo_Stato IN ('Attivo', 'Standby', 'Manutenzione',
    'Malfunzionante') THEN
            v_stato_valido := TRUE;
        END IF;

        IF v_stato_valido = FALSE THEN
            DBMS_OUTPUT.PUT_LINE('Errore: lo stato '
    p_Nuovo_Stato  ' non è valido.');
        ELSE
            -- Se il sensore esiste e lo stato è valido, aggiorna
    lo stato
            UPDATE SENSORI
            SET Stato_Operativo = p_Nuovo_Stato
            WHERE ID = p_Sensore_ID;

            DBMS_OUTPUT.PUT_LINE('Stato del sensore '
    p_Sensore_ID  ' aggiornato a '  p_Nuovo_Stato);
        END IF;
    END IF;
END;
```

```
EXECUTE AggiornareStatoSensore(5, 'Manutenzione');
```

Stato del sensore 5 aggiornato a Manutenzione.

```
EXECUTE AggiornareStatoSensore(1, 'Sospeso');
```

Errore: lo stato Sospeso non è valido.

Procedure N.4 Per eseguire un **OPERAZIONE** di *Manutenzione* se *Data_Ultimo_Controllo* è maggiore di 30

```
CREATE OR REPLACE PROCEDURE ControllaManutenzioneSensori AS
    CURSOR membri_cursor IS
        SELECT ID
        FROM MEMBRI
        WHERE Ruolo = 'Manutentore'; -- Trova tutti i manutentori
    v_membro_id INT;
    v_membro_index INT := 0;
    v_tot_membri INT := 0;
    TYPE membri_table_type IS TABLE OF membri_cursor%ROWTYPE INDEX
    BY PLS_INTEGER;
    membri_table membri_table_type;
BEGIN
    -- Inizializza la tabella con i membri
    OPEN membri_cursor;
    LOOP
        FETCH membri_cursor INTO membri_table(v_tot_membri + 1);
        EXIT WHEN membri_cursor%NOTFOUND;
        v_tot_membri := v_tot_membri + 1;
    END LOOP;
    CLOSE membri_cursor;

    -- Verifica la presenza di manutentori, se non ci sono esce
    IF v_tot_membri = 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Nessun manutentore
    disponibile.');
        RETURN;
    END IF;

    -- Verifica per tutti i sensori se la data dell'ultimo
    controllo è più di 30 giorni fa
    FOR sensore_rec IN (SELECT ID, Data_Ultimo_Controllo FROM
    SENSORI) LOOP
        -- Controlla se la data dell'ultimo controllo è più di 30
    giorni fa oppure se è NULL
        IF sensore_rec.Data_Ultimo_Controllo IS NULL OR
    sensore_rec.Data_Ultimo_Controllo < (SYSDATE - 30) THEN
            -- Assegna il prossimo membro disponibile in modo
    ciclico
            v_membro_index := MOD(v_membro_index, v_tot_membri) +
    1;
            v_membro_id := membri_table(v_membro_index).ID;
```

```
34
35              -- Inserisce una nuova operazione di manutenzione
36              INSERT INTO OPERAZIONI (Membro, Sensore, Operazione,
    Data)
37              VALUES (v_membro_id, sensore_rec.ID, 'Manutenzione',
    SYSDATE);
38
39              -- Aggiorna la Data_Ultimo_Controllo del sensore
40              UPDATE SENSORI
41              SET Data_Ultimo_Controllo = SYSDATE
42              WHERE ID = sensore_rec.ID;
43          END IF;
44      END LOOP;
45  END;
```

```
1  EXECUTE ControllaManutenzioneSensori;
```

Nel Capitolo 5.3 approfondiremo come le procedures precedentemente definite siano state integrate nell'applicazione attraverso l'uso di interfacce semplici ed intuitive, progettate per facilitarne l'utilizzo e garantirne l'efficienza operativa.

4.4 Trigger

Trigger N.1 Per aggiornare lo stato di un **SENSORE** in '*Malfunzionante*' in caso di anomalia

```
1  CREATE OR REPLACE TRIGGER aggiorna_stato_sensore
2  AFTER INSERT ON ANOMALIE
3  FOR EACH ROW
4  BEGIN
5      UPDATE SENSORI
6      SET Stato_Operativo = 'Malfunzionante'
7      WHERE ID = :NEW.Sensore;
8  END;
```

Trigger N.2 Per verificare che il valore rilevato da un **SENSORE** sia maggiore di zero, a eccezione dei sensori di temperatura

```
1  CREATE OR REPLACE TRIGGER verifica_valore_rilevato
2  BEFORE INSERT ON RILEVAZIONI
3  FOR EACH ROW
4  DECLARE
5      v_tipo_sensor VARCHAR2(50);
6  BEGIN
7      -- Recupera il tipo del sensore associato alla rilevazione
8      SELECT Tipo
9      INTO v_tipo_sensor
```

```
    FROM SENSORI
    WHERE ID = :NEW.Sensore;

    -- Se il valore e' <= 0 e il sensore non e' di tipo 'Temperatura', solleva un errore
    IF :NEW.Valore <= 0 AND v_tipo_sensor != 'Temperatura' THEN
        RAISE_APPLICATION_ERROR(-20001, 'Il valore rilevato aggiunto e' minore o uguale a 0. Deve essere maggiore di 0 per sensori non di Temperatura.');
    END IF;
END;
```

```
INSERT INTO RILEVAZIONI (ID, DATA, ORA, VALORE, SENSORE) VALUES (9, SYSDATE, SYSTIMESTAMP, -40, 2);
```

```
ORA-20001: Il valore rilevato aggiunto è minore o uguale a 0. Deve essere maggiore di 0 per sensori non di Temperatura.
```

Trigger N.3 Per impostare automaticamente l'attributo *Data_Fine* di una **MISSIONE** quando lo *Stato* di quest'ultima risulta *Completata*

```
CREATE OR REPLACE TRIGGER aggiorna_data_fine_missione
BEFORE UPDATE ON MISSIONI
FOR EACH ROW
BEGIN
    IF :NEW.Stato = 'Completata' AND :OLD.Stato <> 'Completata' THEN
        :NEW.Data_Fine := SYSDATE;
    END IF;
END;
```

Trigger N.4 Per impedire l'inserimento di una **RILEVAZIONE** se il **SENSORE** si trova in *Stato Malfunzionante*

```
CREATE OR REPLACE TRIGGER trg_block_malfunzionante_rilevazioni
BEFORE INSERT ON RILEVAZIONI
FOR EACH ROW
DECLARE
    v_stato_operativo SENSORI.Stato_Operativo%TYPE;
BEGIN
    -- Recupera lo stato operativo del sensore
    SELECT Stato_Operativo
    INTO v_stato_operativo
    FROM SENSORI
    WHERE ID = :NEW.Sensore;

    -- Verifica se lo stato è "Malfunzionante"
    IF v_stato_operativo = 'Malfunzionante' THEN
```

```
        RAISE_APPLICATION_ERROR(-20001, 'Impossibile registrare una rilevazione: il sensore è in stato operativo Malfunzionante.');
    END IF;
END;
```

```
INSERT INTO RILEVAZIONI (ID, Data, Ora, Valore, Sensore) VALUES (5, SYSDATE, SYSTIMESTAMP, 23.5, 4);
```

```
ORA-20001: Impossibile registrare una rilevazione: il sensore è in stato operativo Malfunzionante.
```

4.5 Ruoli

La gestione degli utenti e dei ruoli è un aspetto cruciale nei sistemi di gestione di basi di dati (DBMS), poiché garantisce il controllo degli accessi e la sicurezza delle informazioni. L'assegnazione di permessi specifici agli utenti e ai ruoli permette di definire con precisione chi può accedere a determinati dati e quali operazioni può eseguire su di essi. Questa suddivisione migliora non solo la protezione del database, ma anche la gestione e la manutenzione complessiva del sistema.

Per la corretta realizzazione del progetto sono stati creati due ruoli: *MembroEquipaggio* e *UfficialeDiBordo*.

```
CREATE ROLE MembroEquipaggio;
CREATE ROLE UfficialeDiBordo;
```

4.5.1 Membro dell'Equipaggio

Il ruolo *MembroEquipaggio* è pensato per gli utenti che necessitano di accedere ai dati relativi a missioni, membri, sensori, robot, anomalie, rilevazioni, report, utilizzo di robot e sensori, coinvolgimenti, operazioni e partecipazioni. Questo ruolo è utile per garantire che i membri dell'equipaggio possano consultare le informazioni necessarie senza avere permessi di modifica.

```
GRANT SELECT ON MISSIONI TO MembroEquipaggio;
GRANT SELECT ON MEMBRI TO MembroEquipaggio;
GRANT SELECT ON SENSORI TO MembroEquipaggio;
GRANT SELECT ON ROBOT TO MembroEquipaggio;
GRANT SELECT ON ANOMALIE TO MembroEquipaggio;
GRANT SELECT ON RILEVAZIONI TO MembroEquipaggio;
GRANT SELECT ON REPORT TO MembroEquipaggio;
GRANT SELECT ON UTILIZZO_ROBOT TO MembroEquipaggio;
GRANT SELECT ON UTILIZZO_SENSORI TO MembroEquipaggio;
GRANT SELECT ON COINVOLGIMENTI TO MembroEquipaggio;
GRANT SELECT ON OPERAZIONI TO MembroEquipaggio;
GRANT SELECT ON PARTECIPAZIONI TO MembroEquipaggio;
```

4.5.2 Ufficiale di Bordo

Il ruolo *UfficialeDiBordo* ha i permessi di `SELECT`, `INSERT`, `UPDATE` e `DELETE` su tutte le tabelle. Questo ruolo è pensato per un utente che necessita di accesso completo per gestire e operare su tutte le informazioni relative alle missioni e ai loro componenti.

```
1  GRANT SELECT, INSERT, UPDATE, DELETE ON MISSIONI TO
       UfficialeDiBordo;
2  GRANT SELECT, INSERT, UPDATE, DELETE ON MEMBRI TO UfficialeDiBordo
       ;
3  GRANT SELECT, INSERT, UPDATE, DELETE ON SENSORI TO
       UfficialeDiBordo;
4  GRANT SELECT, INSERT, UPDATE, DELETE ON ROBOT TO UfficialeDiBordo;
5  GRANT SELECT, INSERT, UPDATE, DELETE ON ANOMALIE TO
       UfficialeDiBordo;
6  GRANT SELECT, INSERT, UPDATE, DELETE ON RILEVAZIONI TO
       UfficialeDiBordo;
7  GRANT SELECT, INSERT, UPDATE, DELETE ON REPORT TO UfficialeDiBordo
       ;
8  GRANT SELECT, INSERT, UPDATE, DELETE ON UTILIZZO_ROBOT TO
       UfficialeDiBordo;
9  GRANT SELECT, INSERT, UPDATE, DELETE ON UTILIZZO_SENSORI TO
       UfficialeDiBordo;
10 GRANT SELECT, INSERT, UPDATE, DELETE ON COINVOLGIMENTI TO
       UfficialeDiBordo;
11 GRANT SELECT, INSERT, UPDATE, DELETE ON OPERAZIONI TO
       UfficialeDiBordo;
12 GRANT SELECT, INSERT, UPDATE, DELETE ON PARTECIPAZIONI TO
       UfficialeDiBordo;
```

Come vedremo nel Capitolo 5.6 l'*UfficialeDiBordo* avrà accesso anche ad aree riservate all'interno dell'applicazione a cui il *MembroEquipaggio* non potrà accedere.

N.B.: View, Procedure, Trigger e Ruoli verranno implementati nel DBMS tramite appositi script contenenti il codice: (1) `view_procedure_trigger.sql`, (2) `role.sql`.

5 Oracle APEX

5.1 Introduzione

Oracle APEX (Application Express) è una piattaforma di sviluppo applicativo low-code che consente di creare applicazioni web scalabili, sicure e altamente performanti utilizzando il database Oracle come base.
Con Oracle APEX, è possibile sfruttare strumenti integrati per la creazione di interfacce utente, la gestione dei dati e la personalizzazione delle funzionalità, rendendo il processo di sviluppo rapido ed efficiente. È particolarmente utile per automatizzare processi aziendali, creare report interattivi e implementare soluzioni personalizzate su misura per le esigenze delle organizzazioni.

5.2 Home

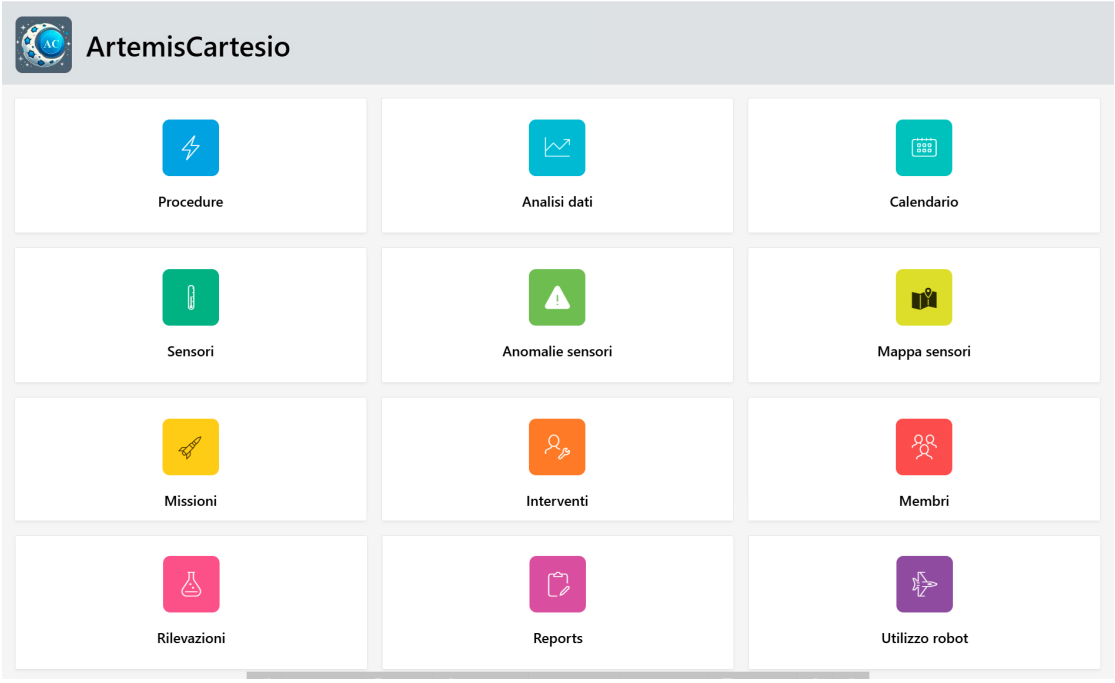


Figure 7: Home page dell'applicazione

In figura (Figura 7) è rappresentata la schermata principale dell'applicazione, progettata per gestire e monitorare ogni aspetto di una missione spaziale. La home page presenta una serie di funzionalità principali, ciascuna accessibile tramite icone intuitive e facilmente identificabili, che semplificano la navigazione e l'interazione con il sistema.

L'interfaccia è user-friendly e ben strutturata, progettata per garantire l'efficienza e la facilità d'uso per i membri dell'equipaggio della missione. Grazie all'organizzazione intuitiva, si può accedere rapidamente alle informazioni necessarie e ai moduli specifici per completare i propri compiti.

5.3 Procedure

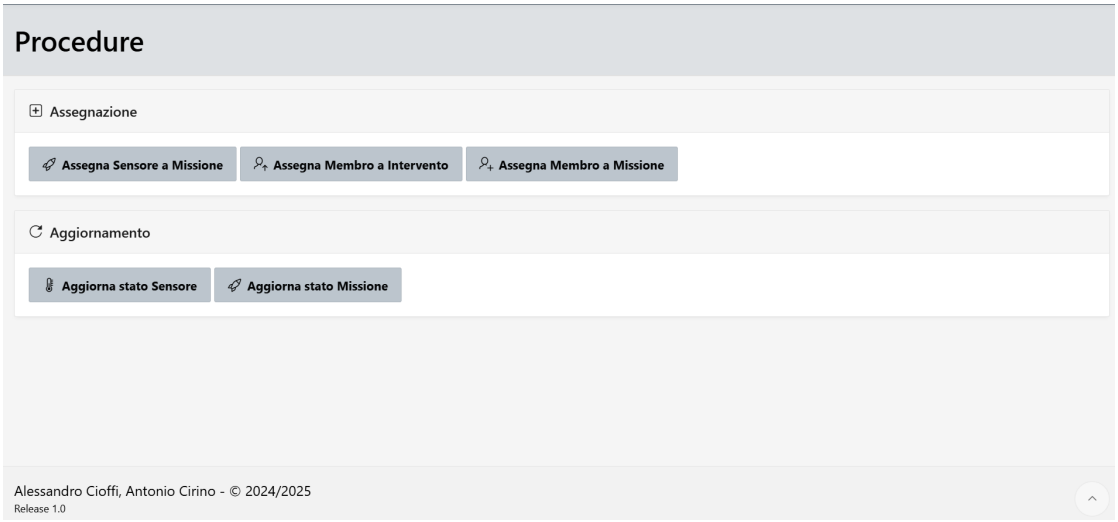


Figure 8: Schermata della sezione *Procedure*

La schermata *Procedure* è divisa in due sezioni: **Assegnazione** e **Aggiornamento**.

Assegnazione Questa sezione permette di gestire l'allocazione delle risorse e del personale per le varie attività. Le funzionalità principali includono:

- **Assegna Sensore a Missione**: consente di associare specifici sensori a una missione attiva.
- **Assegna Membro a Intervento**: permette di designare un membro del team per un determinato intervento tecnico (Figura 9).
- **Assegna Membro a Missione**: utilizzato per assegnare personale a una missione specifica.

Aggiornamento In questa sezione si possono effettuare aggiornamenti sullo stato dei sensori e delle missioni, garantendo che le informazioni siano sempre accurate e aggiornate. Le opzioni includono:

- **Aggiorna stato Sensore**: modifica dello stato operativo di un sensore (attivo, malfunzionante, manutenzione, stand by) e della data dell'ultimo controllo (Figura 10).
- **Aggiorna stato Missione**: consente di aggiornare lo stato di avanzamento di una missione (pianificata, in corso, completata, annullata).

Figure 9: Finestra di dialogo per l'assegnazione di un membro ad un intervento

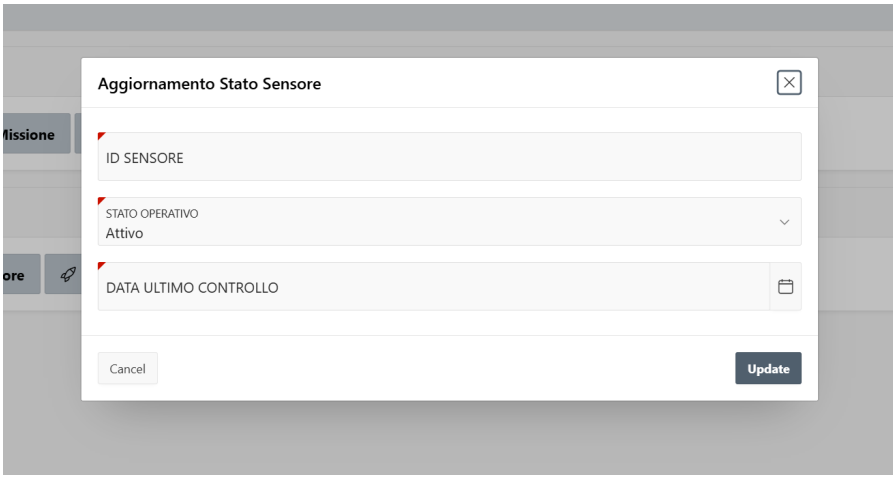


Figure 10: Finestra di dialogo per l'aggiornamento di un sensore

5.4 Analisi dati

L'immagine (Figura 11) mostra la sezione **Analisi Dati** dell'applicazione "Missione Lunare", che fornisce una panoramica visiva sulle principali informazioni relative alla missione. La schermata include diversi grafici, tra cui:

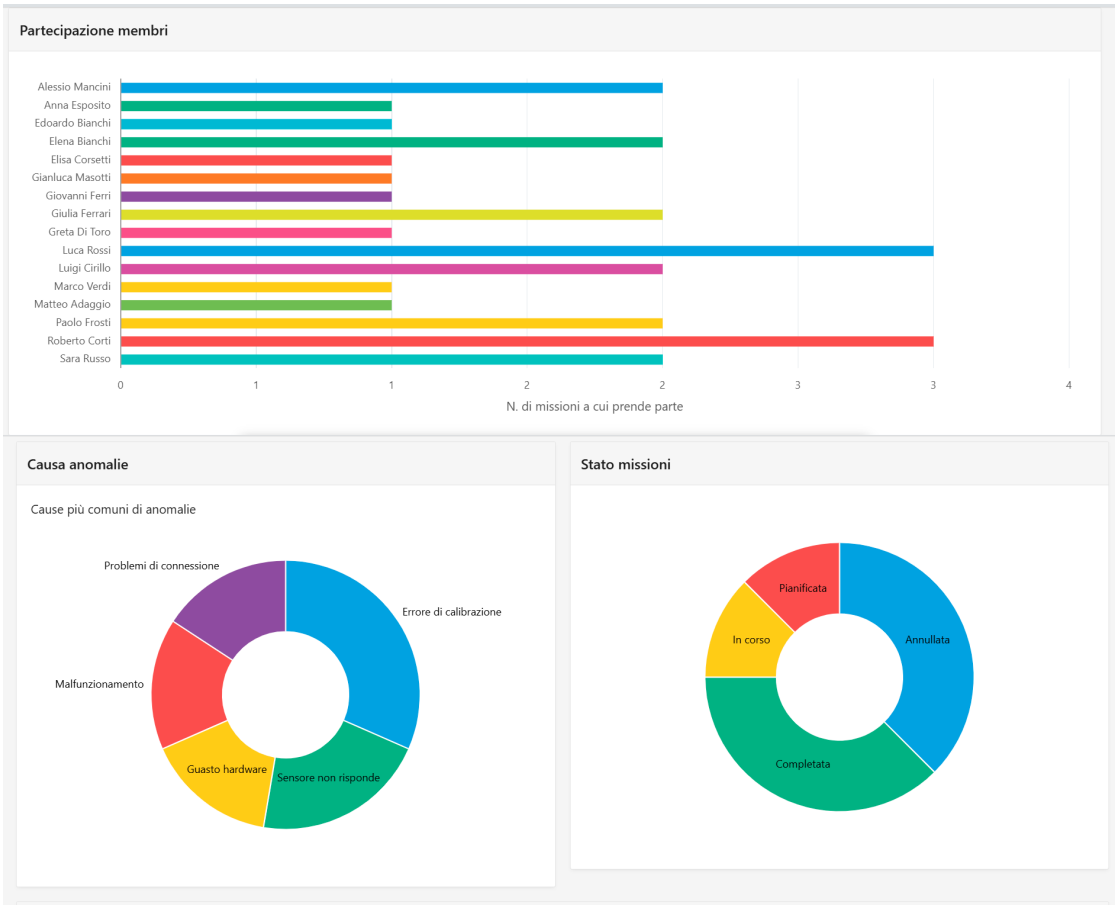


Figure 11: Schermata della sezione *Analisi dati*

Partecipazione membri Un grafico a barre orizzontali che mostra il numero di missioni a cui ciascun membro del team partecipa. Questo consente di visualizzare rapidamente il grado di coinvolgimento dei membri nelle attività.

Causa anomalie Un grafico a ciambella che rappresenta le cause più comuni di anomalie registrate durante le missioni.

Stato missioni Un secondo grafico a ciambella che evidenzia la distribuzione delle missioni in base al loro stato attuale.

Questa sezione offre una visione chiara e intuitiva per monitorare il progresso e le criticità delle missioni, supportando un processo decisionale informato.

5.5 Altre views

Di seguito sono riportate altre sezioni presenti nell'applicazione:

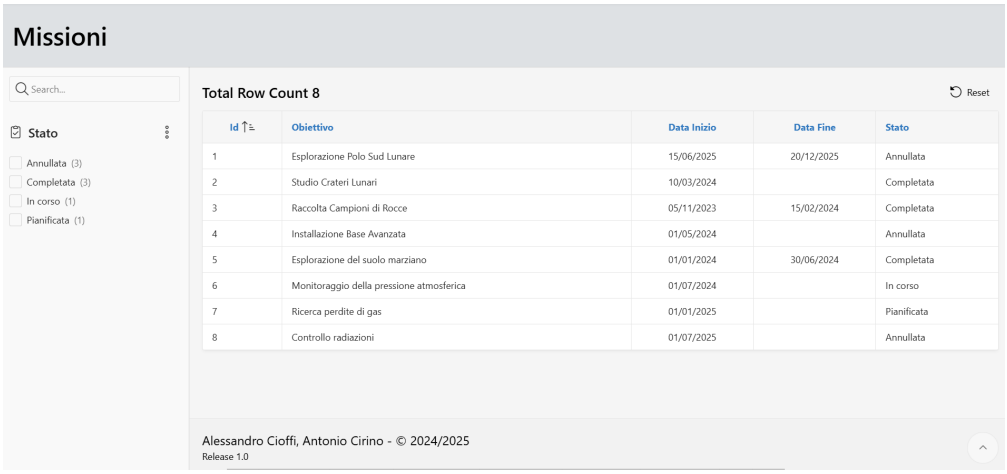


Figure 12: Schermata della sezione *Missioni*

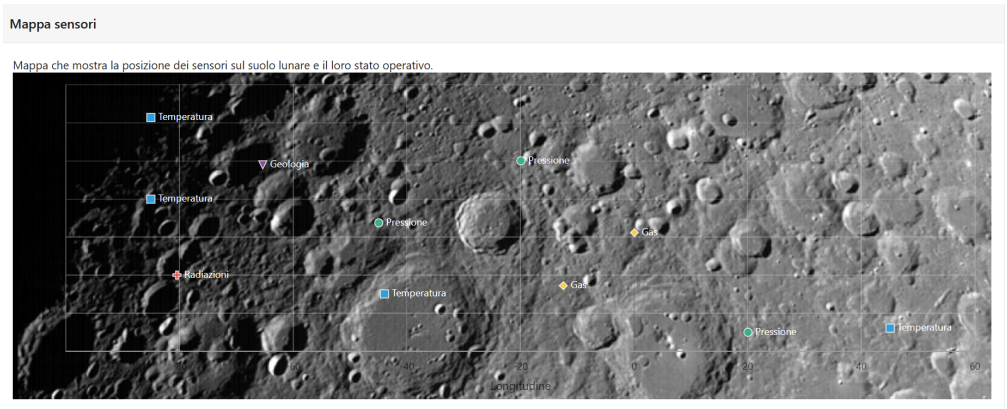


Figure 13: Schermata della sezione *Mappa*

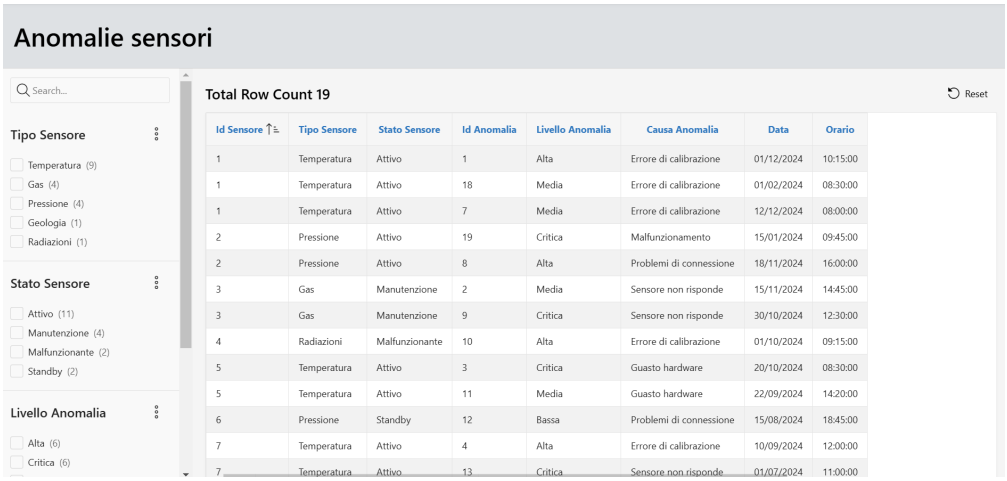


Figure 14: Schermata della sezione *Anomalie*

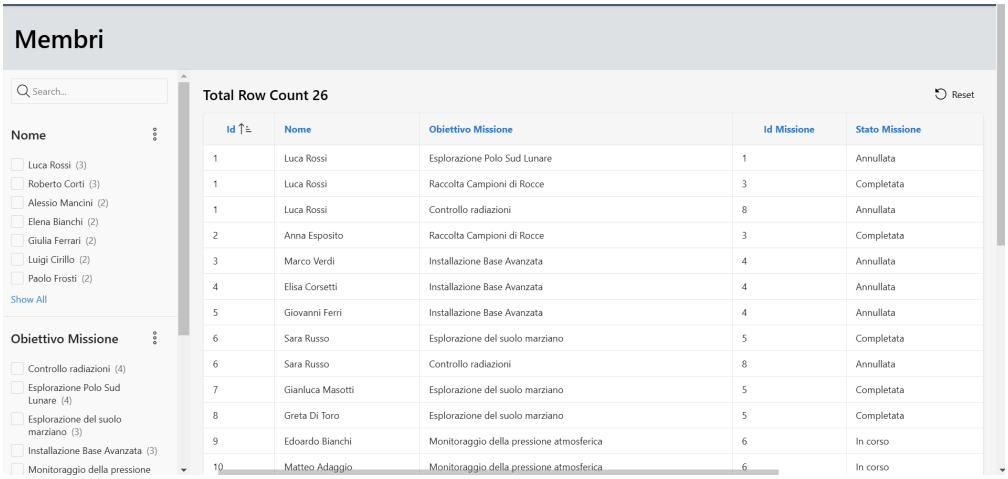


Figure 15: Schermata della sezione *Membri*

5.6 Gestione della sicurezza

Grazie alla sua profonda integrazione con il database Oracle, APEX garantisce sicurezza, affidabilità e un'elevata scalabilità, rendendolo ideale per progetti di qualsiasi dimensione, dalle piccole imprese alle grandi organizzazioni.

L'immagine (Figura 16) mostra la schermata di login dell'applicazione che implementa le best practice per garantire la sicurezza e la protezione dei dati. Questa schermata rappresenta il primo livello di sicurezza dell'applicazione, garantendo che solo utenti autorizzati possano accedere alle funzionalità e ai dati critici del sistema. L'accesso all'applicazione è protetto da un sistema di login che richiede

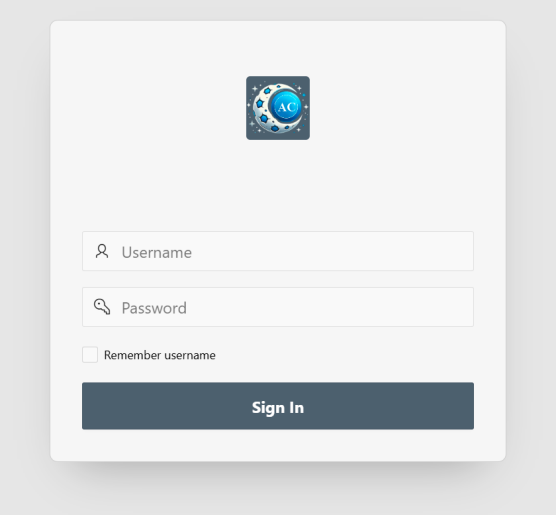


Figure 16: Schermata di login

l'inserimento di username e password.
Oracle APEX utilizza inoltre meccanismi di crittografia per garantire la protezione delle credenziali durante la trasmissione dei dati e il sistema è progettato per prevenire vulnerabilità come SQL injection e brute force.

Inoltre, la schermata Procedure (Figura 8) è accessibile solo agli utenti con ruolo *UfficialeDiBordo* (vedi Capitolo 4.5). Se un utente con ruolo *MembroEquipaggio* provasse ad accedere a questa sezione, andrebbe incontro ad un errore (Figura 17).

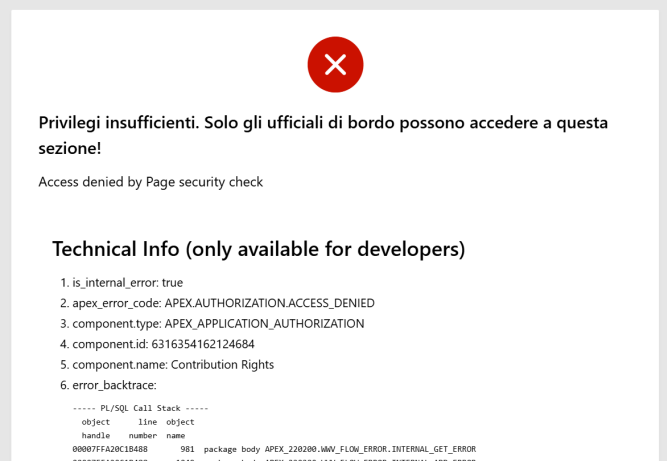


Figure 17: Errore nell'accesso a Procedure

Repository GitHub del Progetto

Il codice sorgente, la documentazione e tutte le risorse relative al progetto sono ospitati in una repository GitHub pubblica. La repository è stata progettata con una struttura chiara e ben organizzata, per semplificare la navigazione e l'utilizzo da parte di studenti, docenti e revisori. Al suo interno è incluso anche il file per il popolamento del database, che non è stato inserito in questa presentazione per evitare di appesantirla inutilmente. Ogni elemento è accompagnato da una documentazione accurata, pensata per garantire un accesso intuitivo e una facile comprensione del progetto.

Link alla Repository

La repository del progetto è disponibile al seguente indirizzo:

`https://github.com/MisterCioffi/ArtemisCartesio`