Informe de Proyecto Final

DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Informe Final de Proyecto

Sistema de Base de Datos para Parques Naturales Argentinos

3 de abril de 2025 Bases de Datos

Integrante	LU	Correo electrónico
Carlos Rafael Giudice	694/15	carlosr.giudice@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón Cero + Infinito)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Conmutador: (+54 11) 5285-9721 / 5285-7400

https://dc.uba.ar

Índice

1. Introducción

Este proyecto busca desarrollar un sistema de base de datos para manejar información sobre parques naturales y áreas protegidas en Argentina. El sistema va a dar herramientas para seguir datos ecológicos, estadísticas de visitantes y actividades de manejo de los parques.

2. Enfoque de Diseño

1. **Test-Driven Development:** Seguimos un enfoque *test-first* para asegurar que todos los requisitos estén claros y se cumplan.

2. **Implementación Modular:** Implementar primero las funcionalidades centrales (*core*), agregando progresivamente más características mientras mantenemos la cobertura de *tests*.

3. Diagrama Entidad-Relación (DER)

Abajo está el DER generado para la base de datos `park_management`, guardado como `pre_computed_results/park_management_er.png`. Muestra cómo las distintas tablas (ej., `parks`, `provinces`, `park_areas`, `natural_elements`, `personnel`, etc.) se relacionan a través de sus *primary keys* y *foreign keys*. Puntos clave incluyen:

- **Relaciones Uno-a-Muchos (*One-to-Many Relationships*)**:
  Por ejemplo, cada parque puede tener múltiples áreas (`park_areas`), y cada área puede albergar múltiples elementos naturales (`area_elements`).

- **Tablas de Especialización (*Specialization Tables*)**:
  `vegetal_elements`, `animal_elements` y `mineral_elements` referencian cada una a `natural_elements` usando una *primary key* compartida, implementando un patrón de herencia de tabla única (*single-table inheritance*) a nivel de base de datos.

- **Tablas de Unión (*Junction Tables*)**:
  Tablas como `area_elements`, `element_food`, `accommodation_excursions` y `visitor_excursions` funcionan como tablas "puente" (*bridge tables*), enlazando relaciones muchos-a-muchos (*many-to-many relationships*).

- **Tabla Relacionada a *Trigger* (*Trigger-Related Table*)**:
  `email_log` se actualiza vía *triggers* cada vez que la cuenta de un elemento disminuye en `area_elements`.

- **Cascadas de *Foreign Key* (*Foreign Key Cascades*)**: La mayoría de las *foreign keys* tienen `ON DELETE CASCADE` u `ON DELETE SET NULL` para asegurar la integridad referencial (*referential integrity*) y una limpieza simplificada.

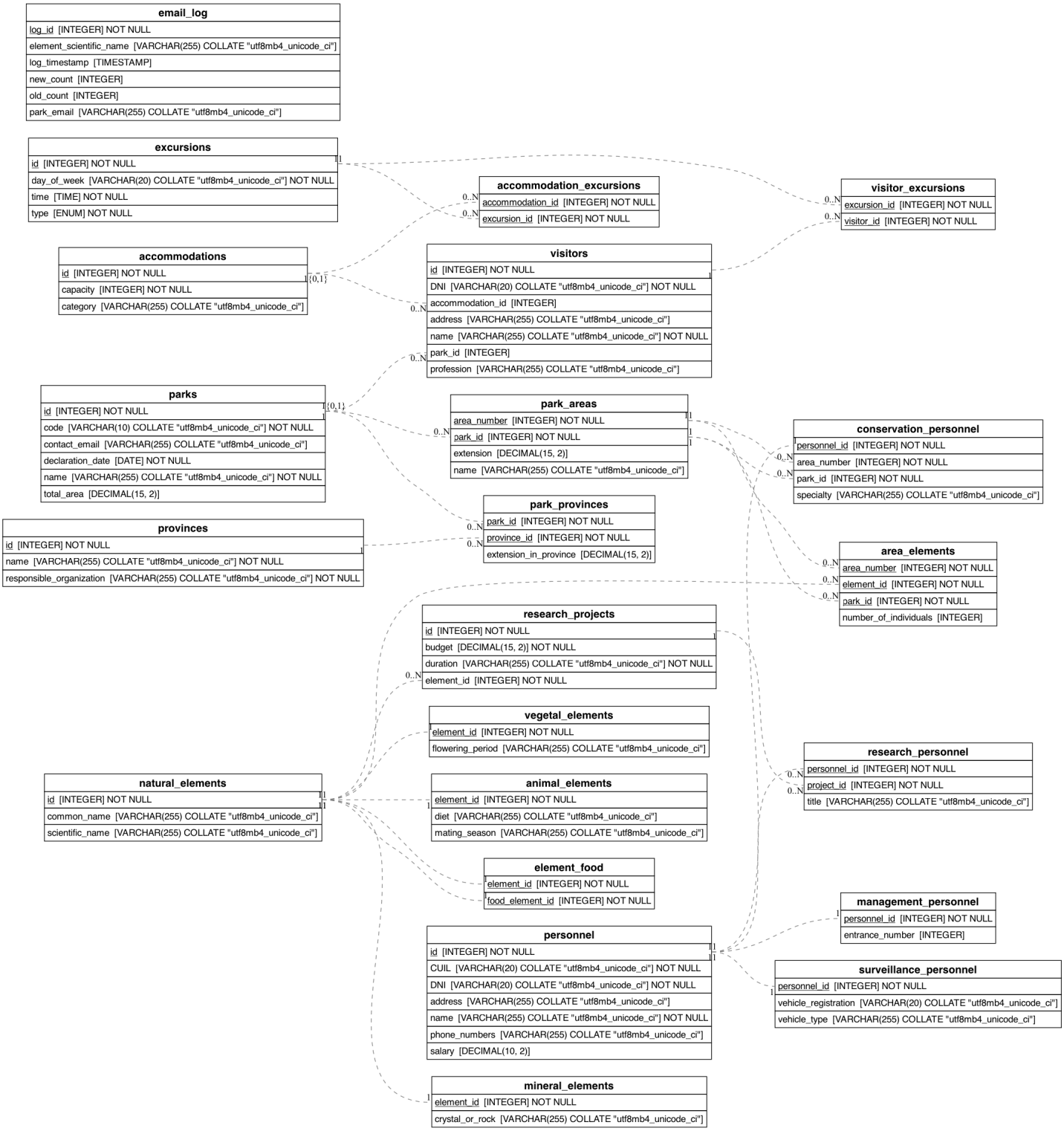

Figura 1: DER de Park Management.

4. Decisiones de Diseño y Simplificaciones

- **Visitantes:** Los visitantes están asociados directamente con el parque que visitan (`visitors.park_id`), aunque se queden en alojamientos. Esto simplifica consultar la cantidad de visitantes por parque.
- **Vínculo con Elemento de Investigación:** Implementado vía la *foreign key* `research_projects.element_id`, asumiendo que un proyecto se enfoca en un elemento principal.
- **Vínculo con Área de Conservación:** Implementado vía la *foreign key* compuesta `conservation_personnel.park_id` y `conservation_personnel.area_number`, asumiendo que un miembro del personal de conservación es asignado a una área específica.
- **Restricciones de Alimento de Elemento (*Element Food Constraints*):** Las restricciones que impiden que los minerales sean alimento (`check_mineral_not_food`) y que las plantas se alimenten (`check_vegetal_not_feeding`) se implementaron usando *triggers* `BEFORE INSERT` y `BEFORE UPDATE` sobre la tabla `element_food`. Se eligió este enfoque porque MySQL no soporta *subqueries* dentro de las *constraints* `CHECK`.

4.1. Propuesta de Índices y Análisis de Plan de Ejecución

4.1.1. Estrategia de Indexación

Los índices están ajustados finamente para soportar las operaciones de *join* y agregación con un *overhead* mínimo.

- **Tabla:** `natural_elements`
  - **Primary Key:** `id`
  - **Unique Constraint:** `scientific_name`
- **Tabla:** `area_elements`
  - **Composite Primary Key:** `(park_id, area_number, element_id)`
  - **Índice Dedicado (*Dedicated Index*):** `idx_area_elements_element_id` sobre `element_id`

Puntos Destacados de Performance (*Performance Highlights*)

- **Joins Eficientes:** El índice dedicado en `area_elements.element_id` permite al *query optimizer* localizar rápidamente las filas (*rows*) que coinciden. La salida del `EXPLAIN` confirma que solo se escanean 136 *rows* usando este índice.

- **Lookups Optimizados:** La *primary key* en `natural_elements.id` soporta un *lookup* `eq_ref`, asegurando una coincidencia uno-a-uno por cada iteración del *join*. Esto garantiza una recuperación rápida de las *rows* necesarias.
- **Agregación Fluida (*Smooth Aggregation*):** El agrupamiento (*grouping*) sobre `natural_elements.id` y `scientific_name` se maneja eficientemente, incluso con las operaciones de *temporary table* y *filesort*. Los índices existentes hacen que el *grouping* y la cláusula `HAVING` (filtrando por un *distinct count* de 1) sean económicos (*cost-effective*).

Análisis del Plan de Ejecución (*Execution Plan Analysis*) Pueden encontrar las explicaciones de los planes de *query* en `pre_computed_results/analysis/execution_plans_output.txt`

```
EXPLAIN SELECT ne.scientific_name
FROM natural_elements ne
JOIN area_elements ae ON ne.id = ae.element_id
GROUP BY ne.id, ne.scientific_name
HAVING COUNT(DISTINCT ae.park_id) = (SELECT COUNT(*) FROM parks);
```

Listing 1: EXPLAIN para Query de Especies en Todos los Parques

El índice sobre `area_elements.element_id` (`idx_area_elements_element_id`) es clave. El `EXPLAIN` muestra que MySQL usa un *nested-loop join*:

- **Lookup en `Area_elements`:** Costo bajo (`read_cost 0.25`, `eval_cost 13.60`). El motor escanea 136 *rows* usando el índice `idx_...`. Accede solo a las columnas indexadas, evitando un *full table scan*.
- **Lookup en `Natural_elements`:** Para cada *row* de `area_elements`, realiza un *lookup* `eq_ref` rápido usando la *primary key*.
- **Agrupamiento y *filesort*:** Agrega costo (`sort_cost 136.00`), minimizado por el uso del índice. MySQL crea una *temporary table* y hace un *filesort*.
- **Subquery `HAVING`:** Eficiente gracias al índice en `parks` (key code).

En general, la elección del índice minimiza los escaneos de *rows* y aprovecha *lookups* rápidos indexados.

```
EXPLAIN SELECT ne.scientific_name
FROM natural_elements ne
JOIN area_elements ae ON ne.id = ae.element_id
GROUP BY ne.id, ne.scientific_name
HAVING COUNT(DISTINCT ae.park_id) = 1;
```

Listing 2: EXPLAIN para Query de Especies en un Solo Parque

El plan muestra a MySQL escaneando 136 *rows* de `area_elements` vía el índice `idx_area_elements_element_id`. Realiza un *lookup* `eq_ref` en `natural_elements`. Agrupa usando *temporary table* y *filesort* (`sort_cost 136.00`). Aplica el filtro `HAVING`. El índice dedicado es óptimo.

4.1.2. Conclusión

Los índices existentes son óptimos para las *queries* actuales, ya que minimizan los escaneos de *rows* y soportan *joins* y agregaciones rápidas y eficientes. Este diseño de indexación robusto no solo mantiene una performance excelente con el *dataset* actual, sino que también es escalable a medida que aumenta el volumen de datos.

4.2. Procedimiento de Comparación de Bases de Datos

Se implementó un *stored procedure* llamado `compare_databases` (en `sql/setup.sql`) que compara esquemas usando *queries* contra el `INFORMATION_SCHEMA`. Acepta dos nombres de bases de datos como parámetros de entrada y revisa diferencias en tablas, estructuras, índices y *constraints*. Es útil para verificar sincronización entre entornos (*sync verification*), comparar estados antes/después de migraciones de esquema (*schema migrations*) y para *troubleshooting* de problemas relacionados a diferencias de esquema.

Uso:

```
-- Llamada de ejemplo:
CALL compare_databases('park_management', 'park_management_backup');
```

Listing 3: Llamada de Ejemplo al Procedimiento compare_databases

4.2.1. Análisis de Resultados de Comparación

La ejecución de `scripts/run_db_comparison.sh` genera la salida de comparación en `results/comparison/schema_comparison_output.txt`. El análisis confirma diferencias intencionales entre los esquemas `park_management` (principal) y `park_management_alt` (alternativo), introducidas en `sql/setup_alternative_db.sql`:

- **Diferencias de Tablas:** `email_log` solo en la principal; `adventure_trails`, `eco_innovations` solo en la alternativa.
- **Diferencias de Índices:** Variaciones en nombres (implícito vs. explícito `idx_...`). Índice `UNIQUE` sobre `provinces.name` falta en la alternativa. Índice `UNIQUE` sobre `natural_elements.common_name` solo en la alternativa. Índice `idx_parks_code` omitido en la alternativa.

- **Diferencias de Constraints:** *Constraint* `UNIQUE` sobre `provinces.name` solo en la principal. FK `visitors_ibfk_2` solo en la principal. *Constraint* `UNIQUE` sobre `natural_elements.common_name` solo en la alternativa.

El procedimiento identifica correctamente las variaciones deliberadas del esquema.

4.3. Control de Concurrencia y Mecanismos de Recuperación

4.3.1. MySQL (InnoDB Engine)

Control de Concurrencia (*Concurrency Control*): Usa *Multi-Version Concurrency Control (MVCC)* para permitir a los lectores (*readers*) acceso *non-blocking* a los datos mientras los escritores (*writers*) los modifican. Implementa *row-level locking* para alta concurrencia. Soporta niveles de aislamiento de transacción estándar (*standard transaction isolation levels*): `READ UNCOMMITTED` (permite *dirty reads*), `READ COMMITTED` (previene *dirty reads*), `REPEATABLE READ` (default, previene *dirty reads* y *non-repeatable reads*), `SERIALIZABLE` (previene todas las anomalías de concurrencia). Usa *gap locks* y *next-key locks* para prevenir *phantom reads* en `REPEATABLE READ`. Detección automática de *deadlocks* (*Deadlock detection*) hace *rollback* de transacciones con menos cambios.

Mecanismos de Recuperación (*Recovery Mechanisms*): Emplea un mecanismo de *write-ahead logging (WAL)* usando *redo logs*. Los cambios se escriben primero al *redo log buffer* y luego se *flushean* a disco. Usa un *doublewrite buffer* para prevenir corrupción de datos por escrituras parciales de página (*partial page writes*). En caso de un *crash*, InnoDB reproduce (*replays*) los *redo logs* desde el último *checkpoint*. Mantiene *undo logs* para *rollback* de transacciones y *MVCC*. Soporta *binary logging* para recuperación a un punto en el tiempo (*point-in-time recovery* - PITR) y replicación (*replication*).

4.3.2. PostgreSQL

Control de Concurrencia (*Concurrency Control*): También usa *MVCC*, dando alta concurrencia entre lectores y escritores. Usa principalmente *row-level locking*. Soporta niveles de aislamiento de transacción estándar: `READ COMMITTED` (default, previene *dirty reads*), `REPEATABLE READ` (previene *dirty reads* y *non-repeatable reads*), `SERIALIZABLE` (previene todas las anomalías de concurrencia). El aislamiento *Serializable* se implementa usando *Serializable Snapshot Isolation (SSI)*. SSI monitorea dependencias de transacciones y aborta transacciones que podrían violar la *serializability*. Comandos de bloqueo explícito (*explicit locking commands*) como `SELECT FOR UPDATE`, `LOCK TABLE` para casos especiales. Detección de *deadlocks* con *timeout* configurable.

Mecanismos de Recuperación (*Recovery Mechanisms*): Usa *Write-Ahead Logging (WAL)*. Los cambios se escriben a archivos de segmento *WAL* antes de que se modifiquen los archivos de datos. Los *Checkpoints* periódicamente *flushean buffers* de datos sucios a disco. En la recuperación, reproduce registros *WAL* desde el último *checkpoint* hacia adelante. Ofrece *Point-in-Time Recovery (PITR)* usando archivado continuo (*continuous archiving*) de registros *WAL*. El archivado *WAL* permite restaurar a cualquier punto en el tiempo. Soporta replicación por *streaming* (*streaming replication*) para alta disponibilidad (*high availability*).

4.3.3. Comparación

Cuadro 1: Comparación de Características de Concurrencia y Recuperación

Feature	**MySQL (InnoDB)**	**PostgreSQL**
Concurrency Model	MVCC	MVCC
Default Isolation	REPEATABLE READ	READ COMMITTED
Phantom Prevention	Gap locks, next-key locks	SSI
Recovery Logging	Redo logs, undo logs	WAL
Point-in-Time Recovery	Binary logs	WAL archiving
Deadlock Handling	Auto detect & rollback	Detect w/ timeout
Locking Granularity	Row-level, table-level	Row-level, table-level
Special Features	Doublewrite buffer	PITR, streaming repl.

Ambos sistemas proveen mecanismos robustos de control de concurrencia y recuperación. PostgreSQL ofrece opciones más avanzadas de *point-in-time recovery* (PITR). MySQL InnoDB puede tener ventajas de performance en algunos escenarios de *workloads OLTP* de alta concurrencia debido a su implementación optimizada de *MVCC*.