Chocolate Doom

Proyecto: Entrega Final

Ivan Santiago Lastra Romero.
Ana Maria Murcia Gomez.
Martin Sanmiguel Delgado.
Lucas Fuentes Sanchez.

Pontificia Universidad Javeriana.

Facultad de Ingeniería.

Bases de Datos.

Profesor: Andrés Oswaldo Calderón.

27 de noviembre de 2025

Índice:

1. Introducción de continuación de proyecto:

Las bases de datos son una de las herramientas más importantes en la Ingeniería de Sistemas, ya que permiten almacenar, organizar y manipular información de manera estructurada, consistente y eficiente. En este campo, el modelo relacional ha sido el más robusto y empleado para asegurar la integridad y consistencia de los datos, al representarlos en tablas enlazadas por medio de claves y restricciones. Además, el lenguaje SQL y los diagramas Entidad-Relación (E-R) ofrecen las herramientas conceptuales y prácticas para modelar, implementar y manipular tales sistemas de datos.

En la primera parte del proyecto Chocolate Doom se modeló conceptual y lógicamente la base de datos a partir de los datos recogidos de una versión alterada del juego Doom. Esta adaptación hizo posible guardar datos de telemetría del jugador en tiempo real (tics), como por ejemplo momentum x,y dirección en x,y,z, ángulo , etc. Además, se estructuró la base para guardar las tablas requeridas bajo el modelo relacional con cada especificación y además el diccionario con datos especificando los datos utilizados

En la segunda parte, se creó físicamente el modelo diseñado, creando la base de datos en PostgreSQL a partir del diagrama E-R normalizado hasta la 3FN, donde se modificó levemente para establecer un modelo más simple contemplando los comentarios hechos dado el tiempo y la relación con los datos. Luego, se importaron a la base de datos los archivos .tsv con más de 23.000 tuplas de telemetría recopiladas en sesiones de juego por los miembros del equipo, donde se repartieron de manera equitativa para establecer un conocimiento del juego y proceder a llenar las tuplas requeridas en la base de datos. Además, se les administró el instrumento PENS a los cuatro participantes reflejando el ámbito de evaluación de UX en la base de datos, cuyos resultados se plasmaron en las tablas correspondientes, respetando las reglas de integridad y relaciones previamente establecidas en el esquema, esto será tratado de forma organizada en el documento de reporte estableciendo paso a paso todos los elementos desarrollados.

Por último, en esta tercera parte, el enfoque se separa más del diseño y creación de la DBS, y por el otro lado, ya con la DBS creada, se analizan y exploran los datos almacenados dentro de la misma. Esto a través de las diferentes indicaciones que nos proveen las instrucciones. Esta entrega prueba la correcta creación y el correcto funcionamiento de la base de datos, principalmente a través de los queries analíticos, al tiempo que aplica principios de optimización y diseño físico del modelo previamente planeados. Esto incluye la creación de índices compuestos y espaciales, la construcción de vistas y vistas materializadas, y el uso de herramientas de automatización mediante Makefile y scripts. De esta forma se concluye el ciclo del proyecto, integrando modelado, creación, estructura relacional, análisis y vinculación

con instrumentos UX, permitiendo explorar patrones de desplazamiento y demás datos telemétricos de los jugadores en Chocolate Doom.

2. Contenido.

En esta entrega se presentan cuatro componentes principales, cada uno orientado a cumplir un conjunto específico de requerimientos funcionales y no funcionales definidos en el enunciado.

Primero, se desarrollan 5 consultas analíticas, las cuales permiten explorar la telemetría registrada de las partidas. Estas consultas corresponden directamente a necesidades como reconstruir trayectorias de un jugador, identificar concurrencia en sectores del mapa, y relacionar la experiencia subjetiva (PENS) con métricas de comportamiento dentro del juego. Todas las consultas están dadas por el enunciado del proyecto, y estas se inspiran en lo desarrollado en el mismo.

Segundo, se presentan tres índices especialmente diseñados para optimizar el acceso a datos de alta frecuencia. Estos incluyen un índice compuesto por XXXXX, un índice XXXXX y un índice XXXXX. Cada índice se justifica mediante su impacto esperado en los planes de ejecución.

Luego, se construyen dos vistas y una vista materializada, orientadas a generar resúmenes reutilizables y acelerar operaciones analíticas frecuentes. Y por último, se incluye un Makefile acompañado de un script de automatización, que permiten la reconstrucción completa de la base de datos: creación del esquema, carga de telemetría, importación de datos desde staging y ejecución de los índices y vistas definidos.

2.1. Queries Analiticas (Analyticas Queries).

A continuación, se realiza una evaluación exhaustiva de la base de datos, analizando las 5 consultas SQL más representativas y evaluables que el equipo participante ha elegido. Esta elección se basa en la capacidad de las queries para obtener métricas y datos relevantes para los objetivos del proyecto Chocolate Doom y su implementación derivada, creada en las entregas #1 y #2. Para cada consulta seleccionada, se define una estructura analítica que incluye: la descripción formal de la información a recuperar; el código SQL a ejecutar; la descripción metodológica de las funciones SQL utilizadas agregación, uniones, subconsultas o SELECTS la evidencia de su resultado mediante una captura de pantalla en el gestor de bases de datos siendo PostgreSQL; y, finalmente, un análisis contextualizado que conecte los resultados obtenidos con el rendimiento y las decisiones de diseño de Chocolate Doom.

Query 1: Distances and trajectories.

En primer lugar, se seleccionó la query #3, cuyo objetivo consiste en identificar la trayectoria más corta y la más larga recorridas por cada jugador dentro de la base de datos. Para construir esta consulta de manera adecuada, fue necesario diseñar una secuencia lógica de pasos que permitiera transformar los datos de telemetría en métricas de distancia significativas. El proceso inicia organizando todos los eventos registrados según el orden temporal de los tics, tanto por partida como por jugador, asegurando así que el análisis preserve la secuencia real del movimiento. Una vez ordenados, se calcula la distancia euclidiana entre la posición del jugador en cada tic y su posición en el tic inmediatamente anterior dentro de la misma partida. Esta operación permite determinar cuánto se desplazó el jugador entre dos registros consecutivos. Posteriormente, se acumulan todas estas distancias parciales para obtener la longitud total de la trayectoria recorrida por cada jugador en cada partida registrada. Con esta

información consolidada, se procede a analizar los valores obtenidos para determinar, por cada jugador, cuál ha sido la menor distancia total recorrida es decir su trayectoria más corta y cuál ha sido la mayor o sea su trayectoria más larga entre todas las partidas que disputó.

A continuación se muestra la consulta que resuelve con los pasos lógicos establecidos la consulta mediante lenguaje SQL.

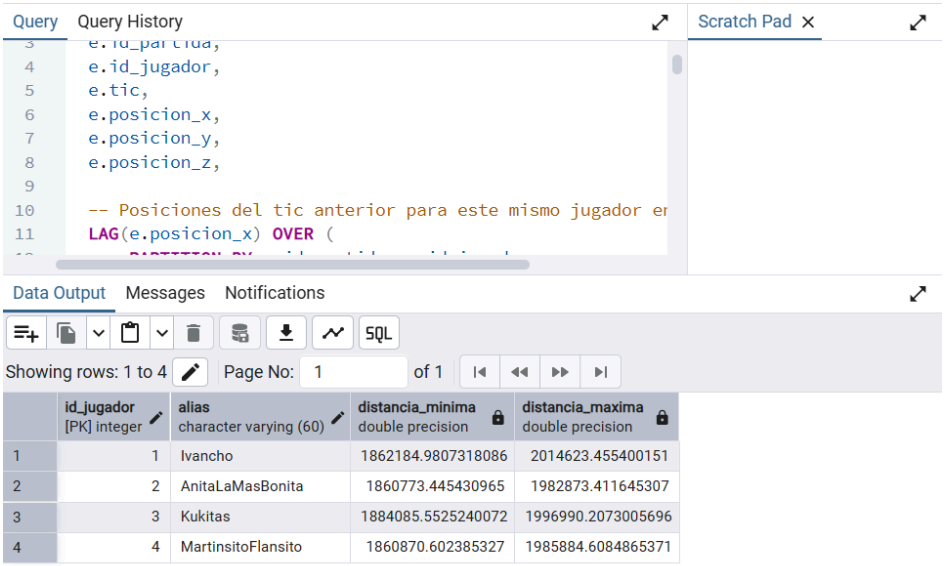
```
WITH eventos_ordenados AS (
    SELECT
        e.id_partida,
        e.id_jugador,
        e.tic,
        e.posicion_x,
        e.posicion_y,
        e.posicion_z,
        -- Posiciones del tic anterior para este mismo jugador en la misma partida
        LAG(e.posicion_x) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS posicion_x_anterior,
        LAG(e.posicion_y) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS posicion_y_anterior,
        LAG(e.posicion_z) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS posicion_z_anterior
    FROM doom.evento_telemetria e
),
distancias_por_trayectoria AS (
    SELECT
        id_partida,
```

```
        id_jugador,

        -- Suma de distancias entre cada tic y su tic anterior
        SUM(
            CASE
                WHEN posicion_x_anterior IS NULL THEN 0.0
                ELSE SQRT(
                    POWER(posicion_x - posicion_x_anterior, 2) +
                    POWER(posicion_y - posicion_y_anterior, 2) +
                    POWER(posicion_z - posicion_z_anterior, 2)
                )
            END
        ) AS distancia_total
    FROM eventos_ordenados
    GROUP BY id_partida, id_jugador
),
extremos_por_jugador AS (
    SELECT
        id_jugador,
        MIN(distancia_total) AS distancia_minima,
        MAX(distancia_total) AS distancia_maxima
    FROM distancias_por_trayectoria
    GROUP BY id_jugador
)
SELECT
    j.id_jugador,
    j.alias,
    e.distancia_minima,
    e.distancia_maxima
```

```
FROM extremos_por_jugador e

JOIN doom.jugador j

  ON j.id_jugador = e.id_jugador

ORDER BY j.id_jugador;
```

La consulta construida hace uso principalmente de funciones ventana, distancia euclidiana y agrupaciones por jugador en tres pasos dentro de subconsultas encadenadas. En un primer momento, la query hace uso de la función ventana LAG() , que permite acceder a la posición del mismo jugador en el tic anterior de la misma partida, asegurando que cada evento se pueda comparar con su historial anterior sin necesidad de un self-join explícito. Con estos datos, la consulta calcula la distancia euclidiana entre los tics sucesivos usando las coordenadas posicion_x, posicion_y y posicion_z. Cada parcial es un pedacito de la trayectoria real del jugador. En la segunda etapa, estas distancias se agregan usando SUM(), sumando todas las distancias entre tics consecutivos para obtener la distancia total recorrida por cada jugador en cada juego. Finalmente, en la tercera etapa, se realiza una última agregación usando las funciones MIN() y MAX() agrupadas por id_jugador para determinar la menor y mayor trayectoria de cada jugador sobre todas sus partidas. La consulta final une estos resultados con la tabla jugador para mostrar también el alias de cada jugador relacionado a su resultado.

Con esto en cuenta, A continuación en la Figura #1 se muestra el resultado obtenido al hacer query #2 listada anteriormente y la descripción anteriormente recalcada en PostgreSQL.



	id_jugador [PK] integer	alias character varying (60)	distancia_minima double precision	distancia_maxima double precision
1	1	Ivancho	1862184.9807318086	2014623.455400151
2	2	AnitaLaMasBonita	1860773.445430965	1982873.411645307
3	3	Kukitas	1884085.5525240072	1996990.2073005696
4	4	MartinsitoFlansito	1860870.602385327	1985884.6084865371

Figura #1: Consulta en PostgreSQL referente a la query trayectorias.

Finalmente se analiza los resultados obtenidos, Los resultados muestran que todos los jugadores tienen trayectorias totales similares, lo que significa patrones de movimiento similares en el juego, pero hay diferencias significativas en los valores mínimos y máximos. Ivancho es el que tiene la mayor distancia recorrida de todas, lo que implica que en una partida exploró mucho más el mapa. Por contra, Kukitas muestra la menor trayectoria

mínima, lo que significa que incluso en su sesión más floja se movió más que el resto en sus sesiones mínimas, demostrando ser muy regular y con mucha movilidad en todas sus sesiones. AnitaLaMasBonita y MartinsitoFlansito se encuentran en un punto medio con valores altos, pero no extremos como los jugadores anteriores, lo que indica un estilo constante de búsqueda pero no más amplio ayudado con el cálculo anterior.

Query 2: UX responses and Trajectories.

Como segunda consulta se usó la query #4, que se apoya en la lógica ya establecida en la query #3 y las rutas ya calculadas para cada jugador. Primero, como en la pregunta anterior, hay que calcular la duración total de la trayectoria por jugador (la suma de todas las distancias entre tics sucesivos de cada partida), usando las diferencias de posiciones y la distancia euclidiana, sumando por id_partida e id_jugador. Luego, de todas esas distancias se saca un promedio general de distancia recorrida que sirve como línea de referencia para medir cómo se comporta cada jugador ante el colectivo. A continuación, se eligen solamente a los jugadores cuya distancia total recorrida (la suma de todas sus partidas) sea superior a ese promedio mundial, ya que son los que tienen sesiones más activas o largas en movimiento por el mapa. Finalmente, estos jugadores filtrados se unen con la tabla de respuesta_ux, para poder listar y analizar las encuestas UX relacionadas solo con aquellos jugadores con trayectorias sobre la media, pero por cada jugador se listan todas las preguntas que tiene respondidas por la tabla respuesta_ux.

A continuación se muestra la consulta que resuelve con los pasos lógicos establecidos la consulta mediante lenguaje SQL.

```
WITH eventos_ordenados AS (

  SELECT

    e.id_partida,

    e.id_jugador,

    e.tic,

    e.posicion_x,

    e.posicion_y,

    e.posicion_z,

    -- Posición en el tic anterior para este mismo jugador

    LAG(e.posicion_x) OVER (

      PARTITION BY e.id_partida, e.id_jugador

      ORDER BY e.tic

    ) AS pos_x_prev,

    LAG(e.posicion_y) OVER (

      PARTITION BY e.id_partida, e.id_jugador

      ORDER BY e.tic
```

```
        ) AS pos_y_prev,
        LAG(e.posicion_z) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS pos_z_prev
    FROM doom.evento_telemetria e
),
trayectorias_por_partida AS (
    SELECT
        id_partida,
        id_jugador,
        SUM(
            CASE
                WHEN pos_x_prev IS NULL THEN 0
                ELSE SQRT(
                    POWER(posicion_x - pos_x_prev, 2) +
                    POWER(posicion_y - pos_y_prev, 2) +
                    POWER(posicion_z - pos_z_prev, 2)
                )
            END
        ) AS distancia_total
    FROM eventos_ordenados
    GROUP BY id_partida, id_jugador
),

distancia_promedio_global AS (
    SELECT AVG(distancia_total) AS promedio_global
    FROM trayectorias_por_partida
),
```

```
jugadores_por_encima_promedio AS (
    SELECT
        t.id_jugador,
        SUM(t.distancia_total) AS distancia_acumulada
    FROM trayectorias_por_partida t
    GROUP BY t.id_jugador
    HAVING SUM(t.distancia_total) >
          (SELECT promedio_global FROM distancia_promedio_global)
)

SELECT
    j.id_jugador,
    j.alias,
    r.id_respuesta,
    r.id_instrumento,
    r.fecha_respuesta,
    r.puntaje_total,
    r.id_pregunta
FROM jugadores_por_encima_promedio jp
JOIN doom.jugador j
     ON j.id_jugador = jp.id_jugador
JOIN doom.respuesta_ux r
     ON r.id_usuario = j.id_usuario
ORDER BY j.id_jugador, r.fecha_respuesta;
```

La consulta se compone de varias subconsultas con WITH que se van encadenando para finalmente hacer un SELECT de las respuestas UX de algunos jugadores. En la primera subconsulta, eventos_ordenados, se parte de la tabla doom.evento_telemetria y se aplican funciones ventana (LAG) particionadas por id_partida e id_jugador y ordenadas por tic para conseguir la posición anterior (pos_x_prev, pos_y_prev, pos_z_prev) de cada jugador en la misma partida sin realizar un self-join. Con esas columnas se tiene la información para calcular distancias entre tics sucesivos. En la segunda subconsulta, trayectorias_por_partida,

se agrupan los datos por id_partida e id_jugador y se usa SUM con CASE: cuando no hay posición anterior como en el caso del primer tic, se suma 0, en caso contrario, se calcula la distancia euclidiana con SQRT y POWER sobre las diferencias de coordenadas. El resultado es distancia_total, la suma de las distancias recorridas por cada jugador en cada partida. En distancia_media_total se calcula un único valor: la media global de todas esas distancias con AVG(distancia_total) que hará de índice.

Luego, en jugadores_por_encima_promedio, se vuelve a agrupar, pero ahora solo por id_jugador, sumando (SUM) la distancia_total de todas sus partidas para sacar distancia_acumulada. Aquí se usa HAVING que filtra tras agrupar para quedarse solo con aquellos jugadores cuya suma de distancias sea superior a la media global calculada en la subconsulta anterior Finalmente, en el SELECT principal se unen las tablas jugadores_por_encima_promedio (jp) y doom.jugador (j) por id_jugador para obtener el alias del jugador, y se hace otro JOIN con doom.respuesta_ux (r) por la condición r.id_usuario = j.id_usuario, recuperando así todas las respuestas UX de los jugadores por encima de la trayectoria promedio.

Con esto en cuenta, A continuación en la Figura #2 y #3 se muestra el resultado obtenido al hacer query #3 listada anteriormente y la descripción anteriormente recalcada en PostgreSQL.

	id_jugador integer	alias character varying (60)	id_respuesta integer	id_instrumento integer	fecha_respuesta timestamp without time zone	puntaje_total numeric (6,2)	id_pregunta integer
14	1	Ivancho	76	1	2025-11-07 19:07:56.526765	5.00	13
15	1	Ivancho	77	1	2025-11-07 19:08:02.886893	3.00	14
16	1	Ivancho	78	1	2025-11-07 19:08:07.645028	4.00	15
17	1	Ivancho	79	1	2025-11-07 19:08:22.515019	3.00	16
18	1	Ivancho	80	1	2025-11-07 19:08:28.712371	5.00	17
19	1	Ivancho	81	1	2025-11-07 19:09:09.526342	6.00	18
20	1	Ivancho	82	1	2025-11-07 19:09:53.013087	6.00	18
21	1	Ivancho	83	1	2025-11-07 19:10:00.962639	7.00	19
22	1	Ivancho	84	1	2025-11-07 19:10:08.290417	5.00	20
23	1	Ivancho	85	1	2025-11-07 19:10:14.357778	6.00	21
24	2	AnitaLaMasBonita	11	1	2025-11-07 18:22:04.271287	6.00	1
25	2	AnitaLaMasBonita	14	1	2025-11-07 18:31:11.109363	6.00	2
26	2	AnitaLaMasBonita	17	1	2025-11-07 18:31:34.43745	7.00	3
27	2	AnitaLaMasBonita	18	1	2025-11-07 18:31:45.94344	5.00	4
28	2	AnitaLaMasBonita	19	1	2025-11-07 18:31:57.409582	4.00	5
29	2	AnitaLaMasBonita	20	1	2025-11-07 18:32:21.542439	5.00	6
30	2	AnitaLaMasBonita	21	1	2025-11-07 18:32:31.382747	6.00	7
31	2	AnitaLaMasBonita	22	1	2025-11-07 18:32:40.573544	5.00	8
32	2	AnitaLaMasBonita	25	1	2025-11-07 18:33:08.366339	7.00	9
33	2	AnitaLaMasBonita	26	1	2025-11-07 18:33:20.750134	7.00	10
34	2	AnitaLaMasBonita	27	1	2025-11-07 18:33:30.129066	7.00	11
35	2	AnitaLaMasBonita	28	1	2025-11-07 18:33:39.029198	6.00	12

Figura #2: Primera parte de resultados de la query #3.

54	3	Kukitas	49	1	2025-11-07 18:43:16.05929	7.00	10
55	3	Kukitas	50	1	2025-11-07 18:43:22.887868	6.00	11
56	3	Kukitas	51	1	2025-11-07 18:43:29.64174	5.00	12
57	3	Kukitas	52	1	2025-11-07 18:43:48.986791	5.00	12
58	3	Kukitas	53	1	2025-11-07 18:44:05.379637	5.00	13
59	3	Kukitas	54	1	2025-11-07 18:44:18.844198	2.00	14
60	3	Kukitas	55	1	2025-11-07 18:44:22.912598	2.00	15
61	3	Kukitas	56	1	2025-11-07 18:44:30.675747	2.00	16
62	3	Kukitas	57	1	2025-11-07 18:45:14.327266	2.00	16
63	3	Kukitas	58	1	2025-11-07 18:45:36.87478	5.00	17
64	3	Kukitas	59	1	2025-11-07 18:45:53.964076	3.00	18
65	3	Kukitas	60	1	2025-11-07 18:46:13.691757	1.00	19
66	3	Kukitas	61	1	2025-11-07 18:46:18.348717	1.00	20
67	3	Kukitas	62	1	2025-11-07 18:46:22.450922	1.00	21
68	4	MartinsitoFlansito	86	1	2025-11-07 19:13:39.994667	7.00	1
69	4	MartinsitoFlansito	87	1	2025-11-07 19:13:42.768513	7.00	2
70	4	MartinsitoFlansito	88	1	2025-11-07 19:13:45.719478	7.00	3
71	4	MartinsitoFlansito	89	1	2025-11-07 19:13:48.115519	7.00	4
72	4	MartinsitoFlansito	90	1	2025-11-07 19:13:49.583088	7.00	5
73	4	MartinsitoFlansito	91	1	2025-11-07 19:13:50.644192	7.00	6
74	4	MartinsitoFlansito	92	1	2025-11-07 19:14:03.48048	4.00	7

Figura #3: Segunda parte de resultados correspondientes a la query #3.

Los resultados revelan una variedad de respuestas UX para los cuatro jugadores del sistema: Ivancho, AnitaLaMasBonita, Kukitas y MartinsitoFlansito. Esto se debe a que, según el cálculo previo de trayectorias, los cuatro jugadores superan la media global de distancia total recorrida, por lo que todos ellos tienen un patrón de movimiento muy por encima de la media de las partidas. Es decir, que no existió ningún jugador con trayectorias globales por debajo del promedio, y por lo tanto todos son catalogados como jugadores con trayectorias por encima del promedio en el criterio establecido por la consulta.

Por lo tanto, la consulta devuelve todas las respuestas UX relacionadas con esos jugadores, lo cual es esperado porque la tabla respuesta_ux guarda cada pregunta respondida como un registro separado. Por eso existen varias filas por jugador, cada una siendo un ítem del instrumento UX con su respectiva id_respuesta, id_pregunta, puntaje_total y fecha_respuesta. Este patrón indica que los jugadores han realizado un cuestionario UX con varias preguntas, por eso hay decenas de respuestas por jugador: el sistema está mostrando toda la traza UX completa de los jugadores que han alcanzado trayectorias altas.

Query 3: Who has Minor trajectory?

Para crear esta consulta que corresponde a la query #7 se usó un proceso para poder vincular las rutas de movimiento con las respuestas de experiencia de usuario. Primero, se ordenaron los eventos de telemetría por jugador y por partida, creando una función ventana (LAG) para recuperar la posición del tic anterior, información necesaria para calcular la distancia euclidiana recorrida entre tics sucesivos. Luego, estas distancias se sumaron por episodio y por jugador para tener la trayectoria total de cada jugador en cada episodio del juego. Una vez calculadas las rutas, se determinó para cada episodio la menor distancia encontrada con una ventana particionada por id_episodio para poder compararlas entre sí. Con ese valor mínimo se coló al jugador o jugadores que realmente menos distancia recorrieron en su episodio, conformando así el grupo de jugadores con menores distancias recorridas. Al final, estos jugadores se conectaron con las tablas jugador, usuario y respuesta_ux para poder recuperar todas sus respuestas UX y calcular el promedio de sus puntuaciones.

A continuación se muestra la consulta que resuelve con los pasos lógicos establecidos la consulta mediante lenguaje SQL.

```
WITH eventos_ordenados AS (
    SELECT
        j.id_episodio,
        e.id_partida,
        e.id_jugador,
        e.tic,
        e.posicion_x,
        e.posicion_y,
        e.posicion_z,
        LAG(e.posicion_x) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS pos_x_prev,
        LAG(e.posicion_y) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS pos_y_prev,
        LAG(e.posicion_z) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS pos_z_prev
    FROM doom.evento_telemetria e
    JOIN doom.juego j
      ON j.id_partida = e.id_partida
),
trayectorias_por_episodio AS (
    SELECT
        id_episodio,
```

```
        id_jugador,
        SUM(
          CASE
            WHEN pos_x_prev IS NULL THEN 0
            ELSE SQRT(
              POWER(posicion_x - pos_x_prev, 2) +
              POWER(posicion_y - pos_y_prev, 2) +
              POWER(posicion_z - pos_z_prev, 2)
            )
          END
        ) AS distancia_total
    FROM eventos_ordenados
    GROUP BY id_episodio, id_jugador
),
trayectorias_minimas AS (
    SELECT
        id_episodio,
        id_jugador,
        distancia_total,
        MIN(distancia_total) OVER (PARTITION BY id_episodio) AS distancia_minima_en_episodio
    FROM trayectorias_por_episodio
),
jugadores_mas_cortos AS (
    SELECT
        id_episodio,
        id_jugador,
        distancia_total
    FROM trayectorias_minimas
    WHERE distancia_total = distancia_minima_en_episodio
```

```
)
SELECT
    jm.id_episodio,
    ep.nombre_episodio,
    j.id_jugador,
    j.alias,
    AVG(r.puntaje_total) AS puntaje_ux_promedio
FROM jugadores_mas_cortos jm
JOIN doom.episodio ep
    ON ep.id_episodio = jm.id_episodio
JOIN doom.jugador j
    ON j.id_jugador = jm.id_jugador
JOIN doom.usuario u
    ON u.id_usuario = j.id_usuario
JOIN doom.respuesta_ux r
    ON r.id_usuario = u.id_usuario
GROUP BY
    jm.id_episodio,
    ep.nombre_episodio,
    j.id_jugador,
    j.alias
ORDER BY
    jm.id_episodio,
    j.id_jugador;
```

La consulta se compone de varias subconsultas WITH encadenadas hasta el SELECT final. En la primera subconsulta, eventos_ordenados, se hace un join de la tabla doom.evento_telemetria con doom.juego para obtener el id_episodio, y se crean funciones de ventana LAG() particionadas por id_partida e id_jugador y ordenadas por tic; esto crea las columnas pos_x_prev, pos_y_prev y pos_z_prev, que son las posiciones anteriores del mismo jugador en la misma partida, lo que permite compararlas con su tic anterior. En la segunda subconsulta, trayectorias_por_episodio, se usa la función de agregación SUM() con un CASE para ir sumando las distancias: cuando las posiciones anteriores son nulas (primer tic) se suma 0; en caso contrario, se calcula la distancia euclidiana con SQRT() y POWER() sobre las

diferencias entre las posiciones actuales y las anteriores; se agrupa (GROUP BY) por id_episodio e id_jugador, guardándose como distancia_total. En la tercera subconsulta, trayectorias_minimas, se conserva la misma relación con episodio a jugador , pero se agrega una columna calculada con MIN(distancia_total) OVER para obtener la distancia mínima por episodio (distancia_minima_en_episodio) sin perder el detalle por jugador. De ahí en adelante, la subconsulta jugadores_mas_cortos hace un filtro con un WHERE distancia_total = distancia_minima_en_episodio, quedándose solo con los jugadores cuya trayectoria sea la mínima de su episodio. En el SELECT final, se coge esta subconsulta y se hace JOIN: con doom.episodio para sacar nombre_episodio, con doom.jugador para sacar alias y con doom.usuario y doom.respuesta_ux para relacionar las rutas con las encuestas UX. Finalmente, se aplica AVG(r.puntaje_total) como función de agregación para obtener el promedio de puntaje UX por jugador con trayectoria mínima por episodio, agrupando (GROUP BY) por jm.id_episodio, ep.nombre_episodio, j.id_jugador y j.alias.

Con esto, A continuación en la Figura #4 se muestra el resultado obtenido al hacer query #7 listada anteriormente y la descripción anteriormente recalcada en PostgreSQL.

```
WITH eventos_ordenados AS (
    SELECT
        j.id_episodio,
        e.id_partida,
        e.id_jugador,
        e.tic,
        e.posicion_x,
        e.posicion_y,
        e.posicion_z,

        LAG(e.posicion_x) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS pos_x_prev,
        LAG(e.posicion_y) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS pos_y_prev,
```

Data Output | Messages | Notifications

Showing rows: 1 to 1

	id_episodio integer	nombre_episodio character varying (120)	id_jugador integer	alias character varying (60)	puntaje_ux_promedio numeric
1	1	Episode 1	4	MartinsitoFlansito	5.7727272727272727

Figura #4: Resultado obtenido correspondiente a la query #7.

La consulta nos indica que en el Episodio 1 el jugador que menor trayectoria total ha recorrido es el jugador con id_jugador = 4, que tiene un alias de MartinsitoFlansito. Para este jugador, al relacionar sus caminos más cortos con la tabla de encuesta UX y promediar todos sus puntajes, se consiguió un puntaje UX promedio de aproximadamente 5.77 (en una escala donde los valores suelen oscilar entre 1 y 7). Esto quiere decir que, aunque fue el jugador que menos se movió en el episodio es decir el que menos distancia recorrió en total en el mapa, su experiencia subjetiva es buena y alta, lo que implica que no hace falta moverse mucho para tener una buena experiencia de juego en ese episodio. Además, la presencia de una única fila significa que en los datos actuales solo hay un episodio (Episodio 1) y un solo jugador con la

trayectoria mínima en ese episodio.

Query 4: Velocity,Distance and duration.

Para crear esta consulta que es equivalente en la query #8, se encadenaron desde los eventos de telemetría con cálculos de tics. En primer lugar, se cogió la tabla evento_telemetria y, con una subconsulta, se ordenaron por partida, jugador y tic, y se hizo un LAG() sobre las columnas de posición (posicion_x, posicion_y, posicion_z) para obtener la posición anterior del mismo jugador en el tic anterior; de esta forma, se pueden comparar puntos de la trayectoria sin tener que hacer self-joins complicados. Con dichas posiciones presentes y pasadas se calcula en la siguiente subconsulta la distancia euclidiana entre tics sucesivos con SQRT() y POWER(), y se suman todas esas distancias con SUM() agrupando solo por id_jugador, obteniendo así la distancia total recorrida por cada jugador en todas sus partidas. En esa misma agregación se calcula también la duración total en tics para cada jugador mediante MAX(tic) - MIN(tic), una medida de cuánto tiempo (en pasos de simulación) ha estado en la telemetría. Finalmente, en el SELECT principal se hace un JOIN con la tabla jugador para obtener el alias y se calcula la velocidad media como la distancia total entre la duración total en tics se tiene en cuenta el caso de duración cero con un CASE, el resultado es una tabla con el id de cada jugador, la distancia total que han recorrido en todo el dataset, el tiempo total que han estado moviéndose y la velocidad media a la que se han desplazado.

A continuación se muestra la consulta que resuelve con los pasos lógicos establecidos la consulta mediante lenguaje SQL.

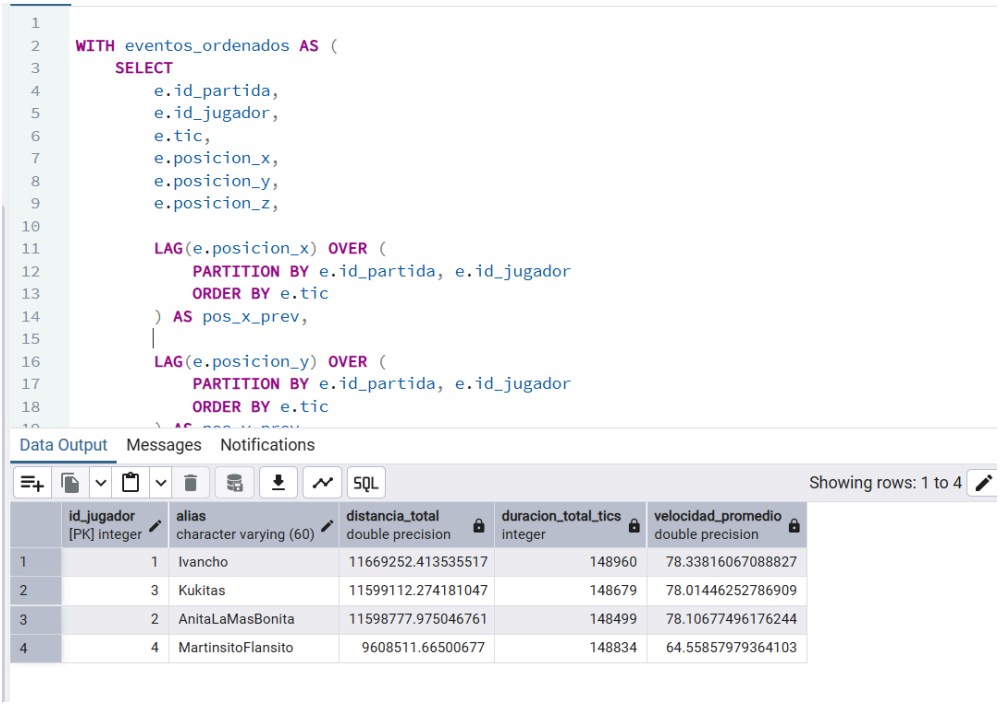
```
WITH eventos_ordenados AS (

  SELECT

    e.id_partida,

    e.id_jugador,

    e.tic,

    e.posicion_x,

    e.posicion_y,

    e.posicion_z,

    LAG(e.posicion_x) OVER (

      PARTITION BY e.id_partida, e.id_jugador

      ORDER BY e.tic

    ) AS pos_x_prev,



    LAG(e.posicion_y) OVER (

      PARTITION BY e.id_partida, e.id_jugador

      ORDER BY e.tic
```

```
        ) AS pos_y_prev,

        LAG(e.posicion_z) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS pos_z_prev
    FROM doom.evento_telemetria e
),

distancia_por_jugador AS (
    SELECT
        id_jugador,

        SUM(
            CASE
                WHEN pos_x_prev IS NULL THEN 0
                ELSE SQRT(
                    POWER(posicion_x - pos_x_prev, 2) +
                    POWER(posicion_y - pos_y_prev, 2) +
                    POWER(posicion_z - pos_z_prev, 2)
                )
            END
        ) AS distancia_total,
        -- duración total en tics para el jugador considerando TODAS sus partidas
        MAX(tic) - MIN(tic) AS duracion_total_tics
    FROM eventos_ordenados
    GROUP BY id_jugador
)
```

```
SELECT
    j.id_jugador,
    j.alias,
    d.distancia_total,
    d.duracion_total_tics,
    CASE
        WHEN d.duracion_total_tics = 0 THEN NULL
        ELSE d.distancia_total / d.duracion_total_tics
    END AS velocidad_promedio
FROM distancia_por_jugador d
JOIN doom.jugador j
  ON j.id_jugador = d.id_jugador
ORDER BY d.distancia_total DESC;
```

La consulta inicial construye una subconsulta en la que se rankean los eventos de telemetría por partida, jugador y tic, usando la función ventana LAG() para obtener la posición anterior del mismo jugador en el tic anterior; con esto se tienen los datos listos para calcular desplazamientos sin usar self-joins. Luego, en una segunda subconsulta, se usan funciones matemáticas como POWER() y SQRT() para calcular la distancia euclidiana entre posiciones sucesivas, y la función de agregación SUM() para sumar todas esas distancias por jugador, obteniendo así la distancia total recorrida. En esa misma etapa se calcula la duración como MAX(tic) - MIN(tic), una métrica agregada que representa el rango completo de actividad del jugador en tics. Finalmente, en el SELECT principal se divide la distancia total entre la duración total para obtener la velocidad promedio donde se usa un CASE para evitar divisiones entre 0, y se hace un JOIN con la tabla jugador para mostrar los datos junto con el alias del jugador, dando como resultado la distancia total, la duración total y la velocidad promedio por jugador.

Con esto, A continuación en la Figura #5 se muestra el resultado obtenido al hacer query #8 listada anteriormente y la descripción anteriormente recalcada en PostgreSQL.



```
WITH eventos_ordenados AS (
    SELECT
        e.id_partida,
        e.id_jugador,
        e.tic,
        e.posicion_x,
        e.posicion_y,
        e.posicion_z,

        LAG(e.posicion_x) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
        ) AS pos_x_prev,

        LAG(e.posicion_y) OVER (
            PARTITION BY e.id_partida, e.id_jugador
            ORDER BY e.tic
```

Data Output | Messages | Notifications

Showing rows: 1 to 4

	id_jugador [PK] integer	alias character varying (60)	distancia_total double precision	duracion_total_tics integer	velocidad_promedio double precision
1	1	Ivancho	11669252.413535517	148960	78.33816067088827
2	3	Kukitas	11599112.274181047	148679	78.01446252786909
3	2	AnitaLaMasBonita	11598777.975046761	148499	78.10677496176244
4	4	MartinsitoFlansito	9608511.66500677	148834	64.55857979364103

Figura #5: Resultado de consulta correspondiente a la query #8.

Los resultados muestran, para cada jugador, la distancia total recorrida en sus partidas, el tiempo total en tics y la velocidad media calculada a partir de estos datos. En primer lugar, Ivancho, Kukitas y AnitaLaMasBonita tienen distancias totales muy parecidas de más de 11.6 millones de unidades, lo que nos dice que son jugadores que recorrieron grandes distancias en el mapa. Además, su tiempo total en tics es casi idéntico alrededor de 148.800 tics, lo que indica que todos estuvieron presentes en el juego durante la misma cantidad de tiempo. Como resultado de estas dos variables, sus velocidades medias son muy similares, en torno a 78 tics/unidad de distancia, lo que indica estilos similares en cuanto a la velocidad de movimiento.

Por otro lado, MartinsitoFlansito si bien su tiempo total de tics es apenas unos cientos menor, su distancia total es mucho menor equivalente a 9.6 millones. Esto resulta en una velocidad media menor, de 64.5, lo que significa que, a pesar de estar casi el mismo tiempo que el resto en movimiento, recorre menos distancia y más lento. Esto puede significar un estilo de juego más estacionario, menos exploratorio o más enfocado en áreas pequeñas del mapa.

Query 5: Average Duration per map

Finalmente, se realizó la query #1 que muestra el cálculo de la duración media de las partidas por mapa, en primer lugar, se calculó la duración de cada partida a partir de los tics de telemetría. Para ello, se combinó la tabla juego con evento_telemetria y, para cada id_partida, se determinó el tic mínimo y el tic máximo, cuya diferencia corresponde a la duración de la partida en tics. Luego, cada partida se relaciona con su episodio y, a través de este, con el mapa de la tabla mapa, para que cada duración quedará enlazada con el escenario en el que se jugó. Este proceso se estructuró en una subconsulta llamada duracion_por_partida que retorna la relación partida, mapa, duración. Finalmente, agrupando por mapa y usando la función de agregación AVG() sobre las duraciones, se consiguió la duración media de las sesiones por mapa de toda la base de datos.

A continuación se muestra la consulta que resuelve con los pasos lógicos establecidos la consulta mediante lenguaje SQL.

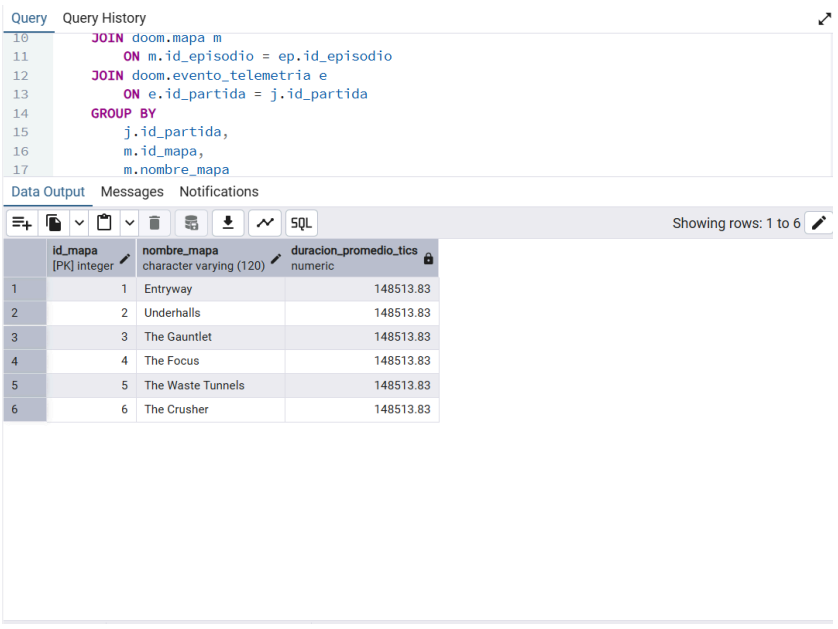
```
WITH duracion_por_partida AS (
    SELECT
        j.id_partida,
        m.id_mapa,
        m.nombre_mapa,
        MAX(e.tic) - MIN(e.tic) AS duracion_tics
    FROM doom.juego j
    JOIN doom.evento_telemetria e
        ON e.id_partida = j.id_partida
    JOIN doom.episodio ep
        ON ep.id_episodio = j.id_episodio
    JOIN doom.mapa m
        ON m.id_episodio = ep.id_episodio
    GROUP BY
        j.id_partida,
        m.id_mapa,
        m.nombre_mapa
)
SELECT
    id_mapa,
    nombre_mapa,
    AVG(duracion_tics) AS duracion_promedio_tics
FROM duracion_por_partida
GROUP BY
    id_mapa,
```

```
    nombre_mapa
ORDER BY
    id_mapa;
```

Explicando la consulta, esta query se compone de dos partes principales. En la primera, la subconsulta duracion_por_partida determina la duración de cada partida: selecciona id_partida, id_mapa, nombre_mapa y usa las funciones agregadas MIN(e.tic) y MAX(e.tic) sobre los datos de telemetría de cada partida. La resta de estos valores, en la forma MAX(e.tic) - MIN(e.tic), se conoce como duracion_tics, que es la duración de la partida en tics. Esta subconsulta hace un NATURAL JOIN entre juego, episodio, mapa y evento_telemetria para relacionar cada juego con su mapa y sus eventos de telemetría.

En la segunda parte, la consulta principal usa como entrada la subconsulta y hace la agregación AVG(duracion_tics) agrupando por id_mapa y nombre_mapa. Esta función saca la media de las duraciones que ya se han sacado de todas las partidas de cada mapa. Finalmente, los datos se ordenan por id_mapa, retornando por cada mapa su código, nombre y duración promedio en tics. Esta estructura hace que siempre se obtenga un valor agregado consistente, incluso si algunas partidas no tienen registros de duración suficientes.

Con esto, A continuación en la Figura #6 se muestra el resultado obtenido al hacer query #1 listada anteriormente y la descripción anteriormente recalcada en PostgreSQL.



	id_mapa [PK] integer	nombre_mapa character varying (120)	duracion_promedio_tics numeric
1	1	Entryway	148513.83
2	2	Underhalls	148513.83
3	3	The Gauntlet	148513.83
4	4	The Focus	148513.83
5	5	The Waste Tunnels	148513.83
6	6	The Crusher	148513.83

Figura #6: Resultado obtenido de consulta relacionado a la query #1.

Los resultados indican que todos los mapas tienen la misma duración promedio de partida, alrededor de 148 513.83 tics. Esta homogeneidad no es arbitraria, sino que es una propiedad del conjunto de datos. Todas las partidas grabadas tienen en común el mismo suceso y además tienen unos rangos de tics muy parecidos; los valores mínimos y máximos de tics por partida no varían mucho. Como resultado, la duración de cada partida (tic máximo - tic mínimo) da como resultado valores muy similares.

2.2. Índices (Indexes)

En esta subsección se presentarán tres consultas diferentes pero habituales, a las cuales se les aplicarán distintos índices. En cada caso se realizará una comparación del rendimiento antes de crear el índice y después de crearlo, utilizando EXPLAIN ANALYZE sobre la base de datos en PostgreSQL y complementando los resultados con una gráfica comparativa de barras. Asimismo, en el tercer índice se evaluarán diferentes tipos de índices (como HASH y BRIN), que serán comparados frente al índice B-Tree, el tipo de índice por defecto en PostgreSQL. Finalmente, para cada índice se presentará: la consulta utilizada, el resultado de EXPLAIN ANALYZE antes del índice, el resultado después de aplicar el índice y, por último, la comparación y análisis de los tiempos mediante las gráficas de barras.

Índice 1:

La primera consulta elegida para analizar es una consulta común sobre datos de telemetría, en la que se leen los datos de un jugador en una partida específica. La consulta devuelve todas las filas de la tabla doom.evento_telemetria donde id_partida = 1 e id_jugador = 1, trayendo las columnas tic y las coordenadas espaciales (posicion_x, posicion_y, posicion_z). Además, los datos se ordenan de menor a mayor por el campo tic, lo que permite reconstruir la línea de tiempo del jugador en la partida. Esta operación, aunque sencilla en apariencia, puede hacerse costosa cuando la tabla de telemetría tiene miles o millones de registros.

A continuación se presenta la consulta utilizada para este primer índice y la comparación de su rendimiento sin índice y con un índice B-Tree, utilizando el comando EXPLAIN ANALYZE. De este modo, se evalúa la eficiencia de la optimización propuesta a partir del tiempo de ejecución reportado por PostgreSQL, analizando cómo cambia el plan de ejecución y el costo temporal de la consulta antes y después de aplicar el índice.

```
SELECT
    e.id_partida,
    e.id_jugador,
    e.tic,
    e.posicion_x,
    e.posicion_y,
    e.posicion_z
FROM doom.evento_telemetria e
WHERE e.id_partida = 1
  AND e.id_jugador = 1
ORDER BY e.tic;
```

Consulta sin indice:

En la Figura #7 se muestra la ejecución de la consulta descrita sin ningún índice implementado.

```
1   EXPLAIN ANALYZE
2   SELECT
3       e.id_partida,
4       e.id_jugador,
5       e.tic,
6       e.posicion_x,
7       e.posicion_y,
8       e.posicion_z
9   FROM doom.evento_telemetria e
10  WHERE e.id_partida = 1
11    AND e.id_jugador = 1
12  ORDER BY e.tic;
```

Data Output | Messages | Notifications

Showing rows: 1 to 10

	QUERY PLAN text
1	Sort (cost=690.74..691.40 rows=264 width=36) (actual time=2.055..2.095 rows=1010.00 loops=1)
2	Sort Key: tic
3	Sort Method: quicksort Memory: 80kB
4	Buffers: shared hit=332
5	-> Seq Scan on evento_telemetria e (cost=0.00..680.12 rows=264 width=36) (actual time=0.045..1.750 rows=1010.00 loo...
6	Filter: ((id_partida = 1) AND (id_jugador = 1))
7	Rows Removed by Filter: 22198
8	Buffers: shared hit=332
9	Planning Time: 0.257 ms
10	Execution Time: 2.167 ms

Figura #7: Consulta sin índice implementado usando EXPLAIN ANALYSE.

Posteriormente se ejecuta la siguiente Sentencia SQL para crear el índice en las tres tablas respectivas que intervienen en la consulta para evaluar la eficiencia del índice, A continuación se muestran los comandos que intervinieron en la creación de los índices.

CREATE INDEX IF NOT EXISTS idx_evento_telemetria_partida_jugador_tic

ON doom.evento_telemetria (id_partida, id_jugador, tic);

CREATE INDEX IF NOT EXISTS idx_juego_id_episodio

ON doom.juego (id_episodio);

CREATE INDEX IF NOT EXISTS idx_respuesta_ux_id_usuario

ON doom.respuesta_ux (id_usuario);

A continuación tras haber ejecutado las anteriores sentencias para crear los índices se volvió a ejecutar la consulta con el mismo comando EXPLAIN ANALYSE para mirar su eficiencia en la Figura #8 se muestra la ejecución con el índice respectivo.

Figura #8: Ejecución con índice B-tree de PostgreSQL de la consulta respectiva.

Con esto se analiza que hubo un cambio entre la consulta sin a con índice B-tree de PostgreSQL pues mejoro bastante la eficiencia de las tuplas generadas generando en menos tiempo que el que tenía previsto, con esto en cuenta en la Figura #9 se muestra una gráfica de barras comparativa entre ambas.

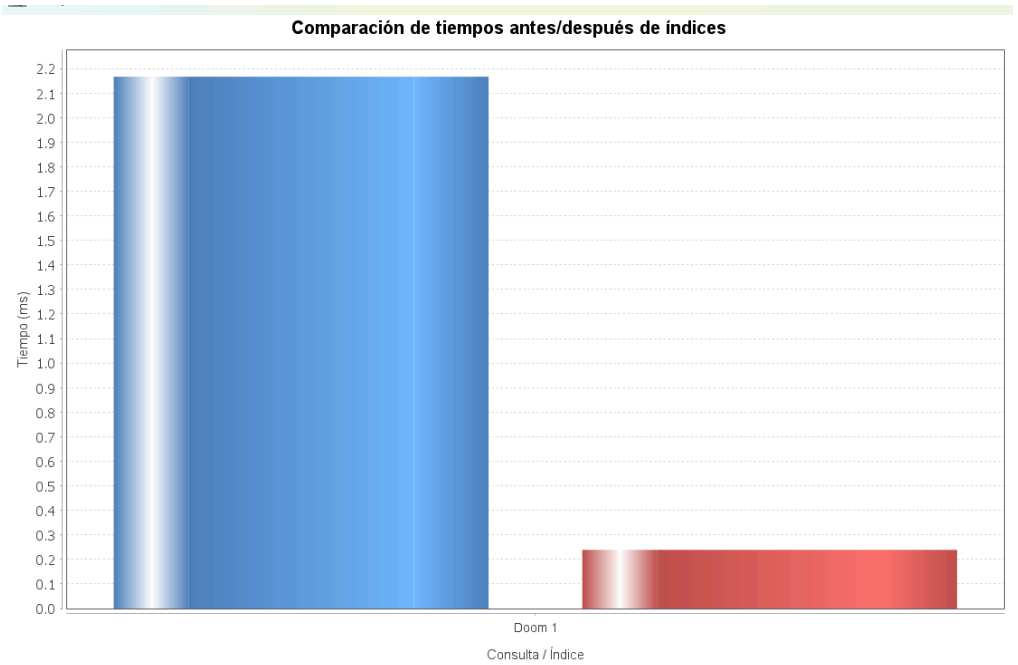


Figura #9: Gráfica de barras comparativa de la primera consulta.

En la gráfica se puede observar claramente cómo la creación del índice B-Tree mejoró la consulta que más se ejecuta, la cual filtra eventos de telemetría por id_partida e id_jugador y luego los ordena por tic. Antes del índice, la consulta tarda alrededor de 2.15 ms en ejecutarse, lo que significa que PostgreSQL tuvo que hacer un sequential scan en la tabla

evento_telemetria, revisando cada fila para encontrar las que coincidieran con las condiciones.

Después de crear el índice compuesto en atributos como id_partida, id_jugador, tic, el tiempo de ejecución se reduce a alrededor de 0.25 ms, una mejora de más del 85–90 %. Esta caída se debe a que PostgreSQL ahora puede hacer un index scan muy selectivo lo que mejora el rendimiento de la consulta en este caso que la query establecida es una común y que puede generar bastantes tuplas.

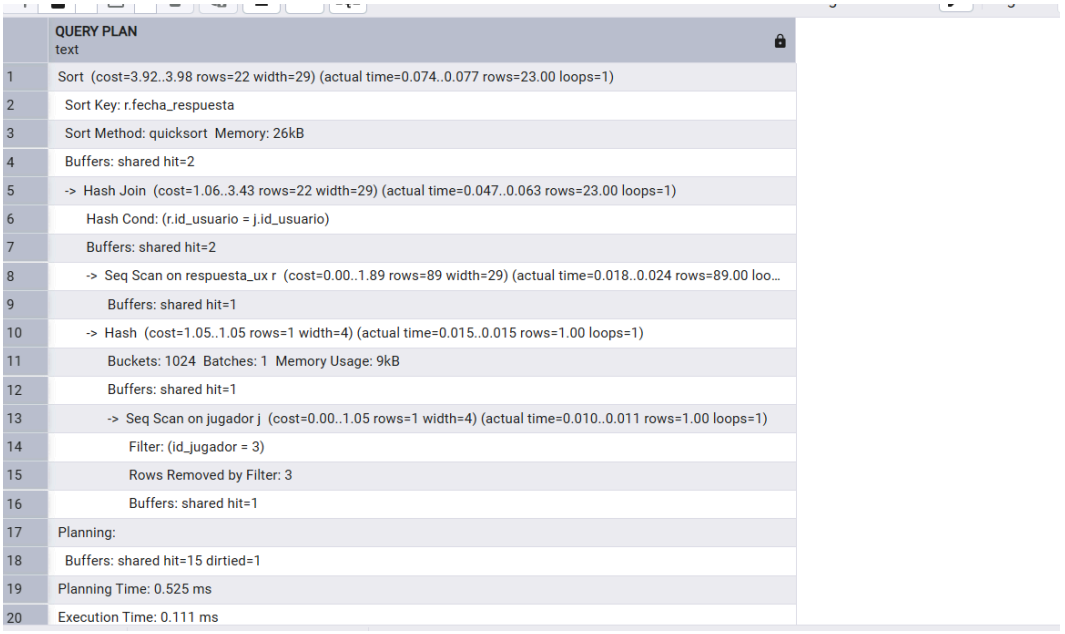
Índice 2:

La segunda consulta elegida para ser analizada con índices, medida con EXPLAIN ANALYZE, es la consulta para obtener las respuestas UX de un jugador en específico. En este caso se manipula la tabla respuesta_ux y se hace un JOIN con la tabla jugador por el campo id_usuario para que cada respuesta quede asociada directamente al jugador que la generó. Esta consulta se eligió porque es un tipo de consulta muy común en sistemas de análisis UX: dado un jugador, obtener todas sus respuestas históricas, ordenadas cronológicamente por fecha de respuesta. Además, aplica filtro por jugador, join y order by time, un caso perfecto para analizar cómo los índices ayudan en la búsqueda por id_usuario y en el ordenamiento por fecha_respuesta.

A continuación se presenta la consulta utilizada para este segundo índice y la comparación de su rendimiento sin índice y con un índice B-Tree, utilizando el comando EXPLAIN ANALYZE. De este modo, se evalúa la eficiencia de la optimización propuesta a partir del tiempo de ejecución reportado por PostgreSQL, analizando cómo cambia el plan de ejecución y el costo temporal de la consulta antes y después de aplicar el índice.

```
EXPLAIN ANALYSE

SELECT

   r.id_respuesta,

   r.id_usuario,

   r.id_instrumento,

   r.fecha_respuesta,

   r.puntaje_total,

   r.id_pregunta

FROM doom.respuesta_ux r

JOIN doom.jugador j

   ON j.id_usuario = r.id_usuario

WHERE j.id_jugador = 3

ORDER BY r.fecha_respuesta;
```

En la Figura #10 se muestra la ejecución de la consulta sin ningun indice implementado con el comando EXPLAIN ANALYSE



	QUERY PLAN text
1	Sort (cost=3.92..3.98 rows=22 width=29) (actual time=0.074..0.077 rows=23.00 loops=1)
2	Sort Key: r.fecha_respuesta
3	Sort Method: quicksort Memory: 26kB
4	Buffers: shared hit=2
5	-> Hash Join (cost=1.06..3.43 rows=22 width=29) (actual time=0.047..0.063 rows=23.00 loops=1)
6	Hash Cond: (r.id_usuario = j.id_usuario)
7	Buffers: shared hit=2
8	-> Seq Scan on respuesta_ux r (cost=0.00..1.89 rows=89 width=29) (actual time=0.018..0.024 rows=89.00 loo...
9	Buffers: shared hit=1
10	-> Hash (cost=1.05..1.05 rows=1 width=4) (actual time=0.015..0.015 rows=1.00 loops=1)
11	Buckets: 1024 Batches: 1 Memory Usage: 9kB
12	Buffers: shared hit=1
13	-> Seq Scan on jugador j (cost=0.00..1.05 rows=1 width=4) (actual time=0.010..0.011 rows=1.00 loops=1)
14	Filter: (id_jugador = 3)
15	Rows Removed by Filter: 3
16	Buffers: shared hit=1
17	Planning:
18	Buffers: shared hit=15 dirtied=1
19	Planning Time: 0.525 ms
20	Execution Time: 0.111 ms

Figura #10: Ejecución de consulta sin índice implementado.

Posteriormente se ejecuta la siguiente Sentencia SQL para crear el índice en las dos tablas respectivas que intervienen en la consulta para evaluar la eficiencia del índice, A continuación se muestran los comandos que intervinieron en la creación de los índices.

CREATE INDEX idx_respuesta_ux_usuario_fecha

ON doom.respuesta_ux (id_usuario, fecha_respuesta);

CREATE INDEX idx_jugador_usuario

ON doom.jugador (id_usuario);

A continuación tras haber ejecutado las anteriores sentencias para crear los índices se volvió a ejecutar la consulta con el mismo comando EXPLAIN ANALYSE para mirar su eficiencia en la Figura #11 se muestra la ejecución con el índice respectivo.

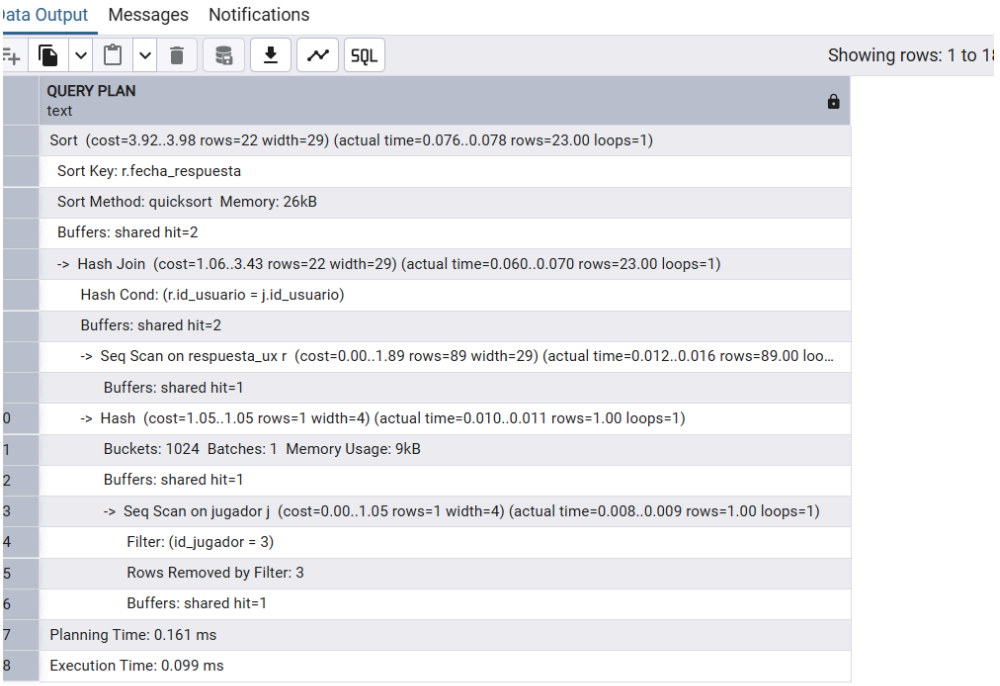


Figura #11: Ejecución de la consulta con B-tree en PostgreSQL.

Con esto se analiza que hubo un cambio entre la consulta sin a con índice B-tree de PostgreSQL pero no muy grande pues apenas difirieron de un par de ms que son insignificantes dada la cantidad de datos que maneja y se muestra que hubo un cambio positivo pero no de una gran magnitud comparada con la anterior, en la Figura #12 se muestra esto comparado en una gráfica de barras.

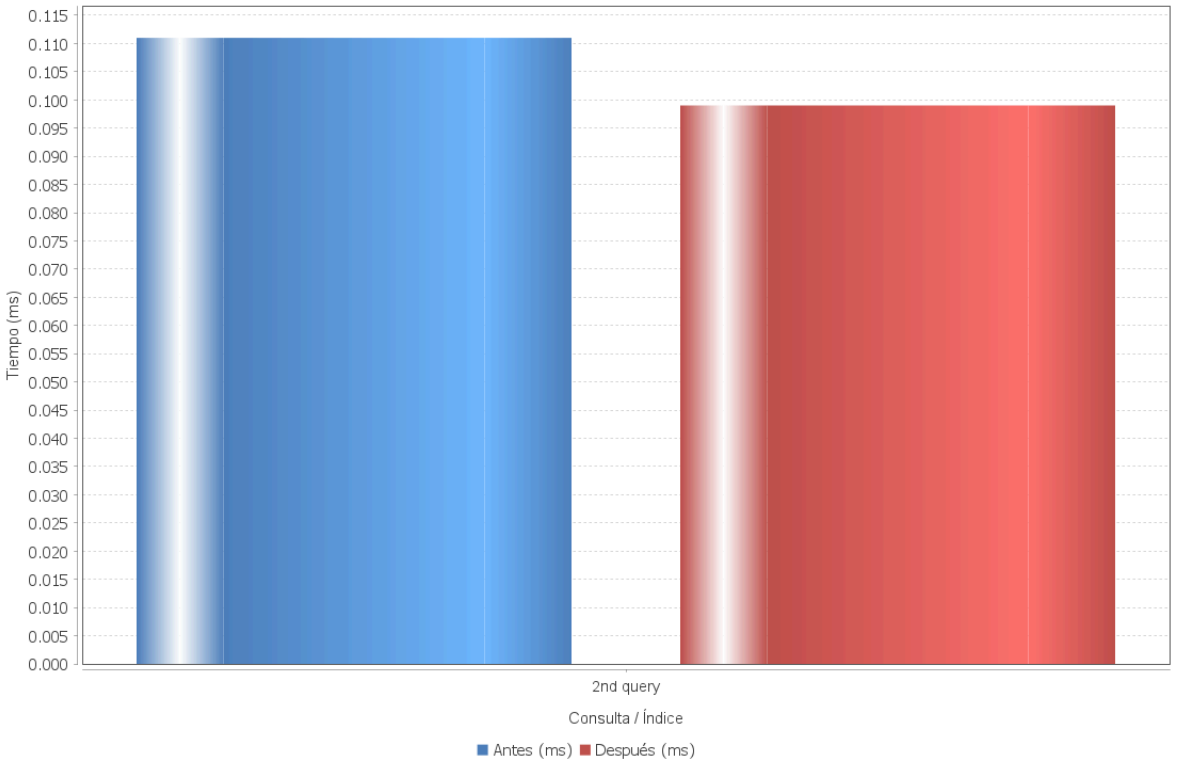


Figura #12: Comparación de rendimiento sin/con índice relacionado a segunda consulta.

En la gráfica se comparan los tiempos de ejecución para la segunda consulta, que recupera las respuestas UX para un jugador realizando un JOIN entre las tablas respuesta_ux y jugador. Los resultados indican que el tiempo antes de crear el índice es de alrededor de 0.110 ms, y después de crear el índice compuesto sobre las columnas (id_usuario, fecha_respuesta), el tiempo baja a alrededor de 0.099 ms.

Si bien la mejora no es tan grande como en la primera consulta, la mejora es constante y demuestra que PostgreSQL puede optimizar la forma en que accede a los datos cuando tiene un índice apropiado.

Índice 3:

La última consulta seleccionada para ser probada con distintos índices, medida con EXPLAIN ANALYZE, es la que obtiene la cantidad total de eventos de telemetría relacionados con una partida específica. En este caso se opera únicamente sobre la tabla evento_telemetria, agrupando por id_partida y agregando con COUNT(*) para contar cuántos registros hay para esa partida. Esta consulta se eligió porque es un patrón común en casos de uso de análisis de telemetría: dado un ID de inicio, calcula rápidamente cuántos eventos se generaron mientras se ejecutaba. Además, al hacer una condición de igualdad sobre una única columna id_partida, es un caso perfecto para comparar el rendimiento de los distintos tipos de índice en PostgreSQL mediante B-Tree, HASH y BRIN y ver cómo influyen en el plan de ejecución y el tiempo de respuesta de la base de datos.

A continuación, la consulta utilizada para este último análisis y comparativa de rendimiento sin índice y con los tres tipos de índices sobre la columna id_partida, B-Tree, HASH y BRIN, medidos con EXPLAIN ANALYZE. Con esta comparación se analiza cómo cada tipo de estructura de indexación influye en el plan de ejecución y el tiempo que tarda en ejecutarse la consulta, mostrando las diferencias entre un índice genérico de propósito general B-Tree, uno optimizado para búsquedas por igualdad HASH y uno para big data con ordenamiento físico BRIN.

```
SELECT

   e.id_partida,

   COUNT(*) AS total_eventos

FROM doom.evento_telemetria e

WHERE e.id_partida = 1

GROUP BY e.id_partida;
```

En la Figura #13 se muestra la ejecución de la consulta con el comando EXPLAIN ANALYSE sin ningún índice asociado

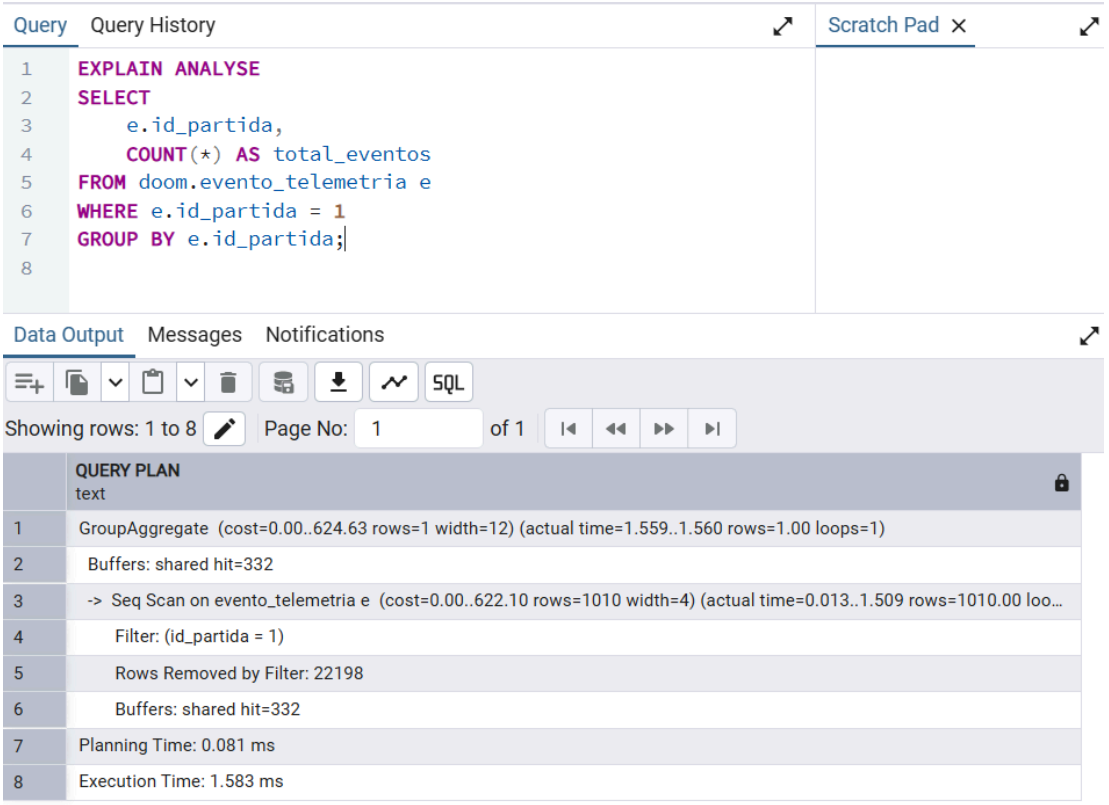


Figura #13: Ejecución de la consulta sin índice asociado.

Posteriormente se ejecuta la siguiente Sentencia SQL para crear el índice en las dos tablas respectivas que intervienen en la consulta para evaluar la eficiencia del índice, A continuación se muestran los comandos que intervinieron en la creación de los índices tanto para B-tree como Hash y BRIN.

CREATE INDEX idx_evento_partida_btree

ON doom.evento_telemetria (id_partida);

CREATE INDEX idx_evento_partida_hash

ON doom.evento_telemetria USING HASH (id_partida);

CREATE INDEX idx_evento_partida_brin

ON doom.evento_telemetria USING BRIN (id_partida);

Con esto en cuenta a continuación en la Figura #14 se muestra el análisis mediante EXPLAIN ANALYSE utilizando el índice de tipo B-tree en PostgreSQL.

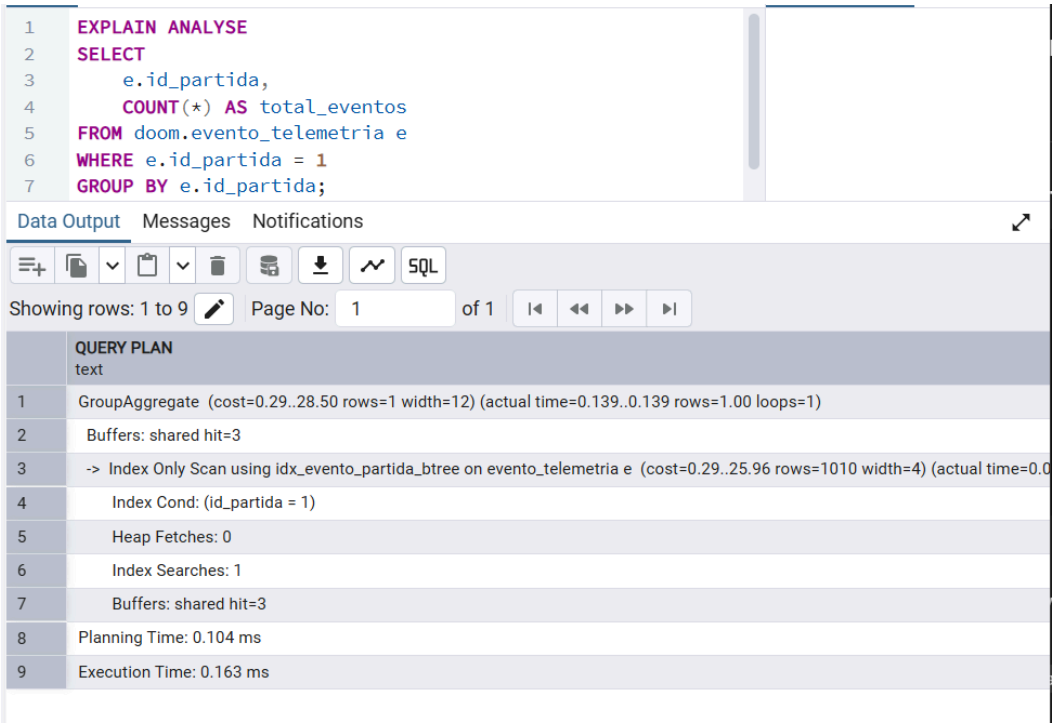


Figura #14: Ejecución de la consulta mediante B-tree en PostgreSQL.

En la Figura #15 se muestra el análisis con el comando EXPLAIN ANALYSE utilizando el índice HASH en PostgreSQL.

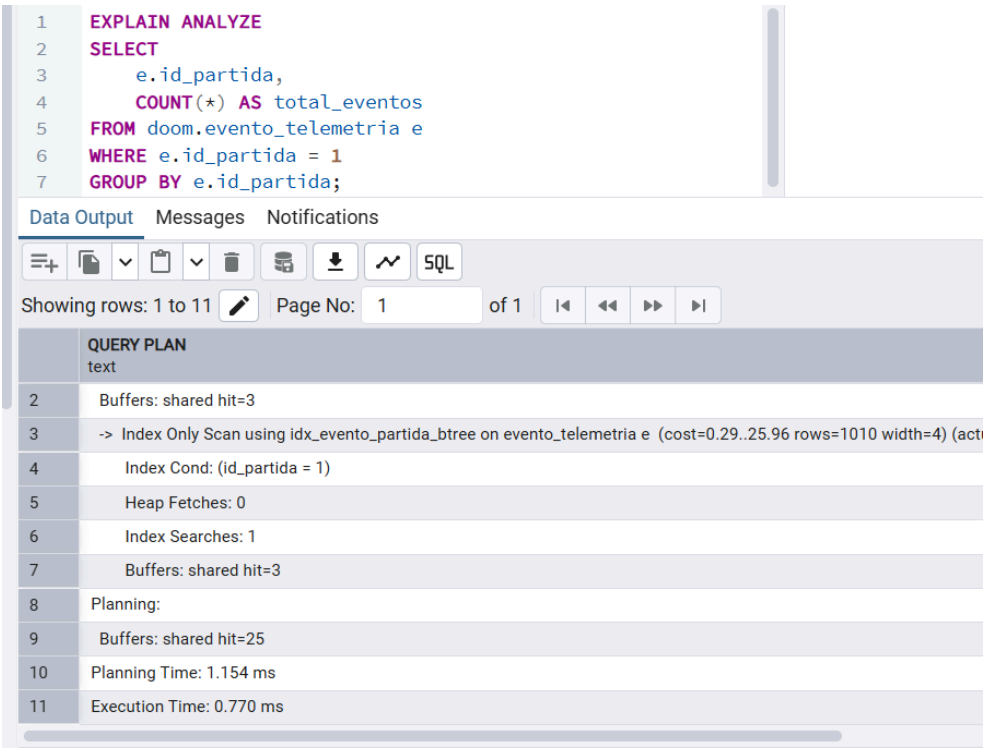


Figura #15: Ejecución de la consulta mediante HASH en PostgreSQL.

En la Figura #16 se muestra el análisis con el comando EXPLAIN ANALYSE utilizando el índice BRIN en PostgreSQL.

Query Query History Scratch Pad

```
1  EXPLAIN ANALYZE
2  SELECT
3      e.id_partida,
4      COUNT(*) AS total_eventos
5  FROM doom.evento_telemetria e
6  WHERE e.id_partida = 1
```

Data Output Messages Notifications

Showing rows: 1 to 11 Page No: 1 of 1

	QUERY PLAN text
1	GroupAggregate (cost=0.29..28.50 rows=1 width=12) (actual time=0.567..0.567 rows=1.00 loops=1)
2	Buffers: shared hit=3
3	-> Index Only Scan using idx_evento_partida_btree on evento_telemetria e (cost=0.29..25.96 rows=1010 width=4) (actual time=0.
4	Index Cond: (id_partida = 1)
5	Heap Fetches: 0
6	Index Searches: 1
7	Buffers: shared hit=3
8	Planning:
9	Buffers: shared hit=27
10	Planning Time: 0.720 ms
11	Execution Time: 0.614 ms

Figura #16: Ejecución de la consulta con BRIN en PostgreSQL.

Con estas pruebas se muestra que el tiempo de ejecución de la consulta cambia en gran medida dependiendo del tipo de índice que se utilice, ya que no todos los índices reaccionan igual ante un filtro de igualdad sobre la columna id_partida. Los resultados demuestran que, si bien el índice HASH puede dar mejoras en ciertas consultas que solo usen igualdades, y el índice BRIN puede ser útil en tablas muy grandes ordenadas físicamente por la columna en cuestión, el índice B-Tree es el más completo. Esto se debe a que B-Tree no solo optimiza la búsqueda por igualdad, sino también los casos de uso mixtos de filtrado, ordenación o agregación, con un rendimiento consistente independientemente de cómo estén distribuidos físicamente los datos.

En la Figura #17 se presenta una comparación entre los tres tipos de índices evaluados B-Tree, HASH y BRIN mostrando el tiempo de ejecución antes y después de aplicar cada uno de ellos.

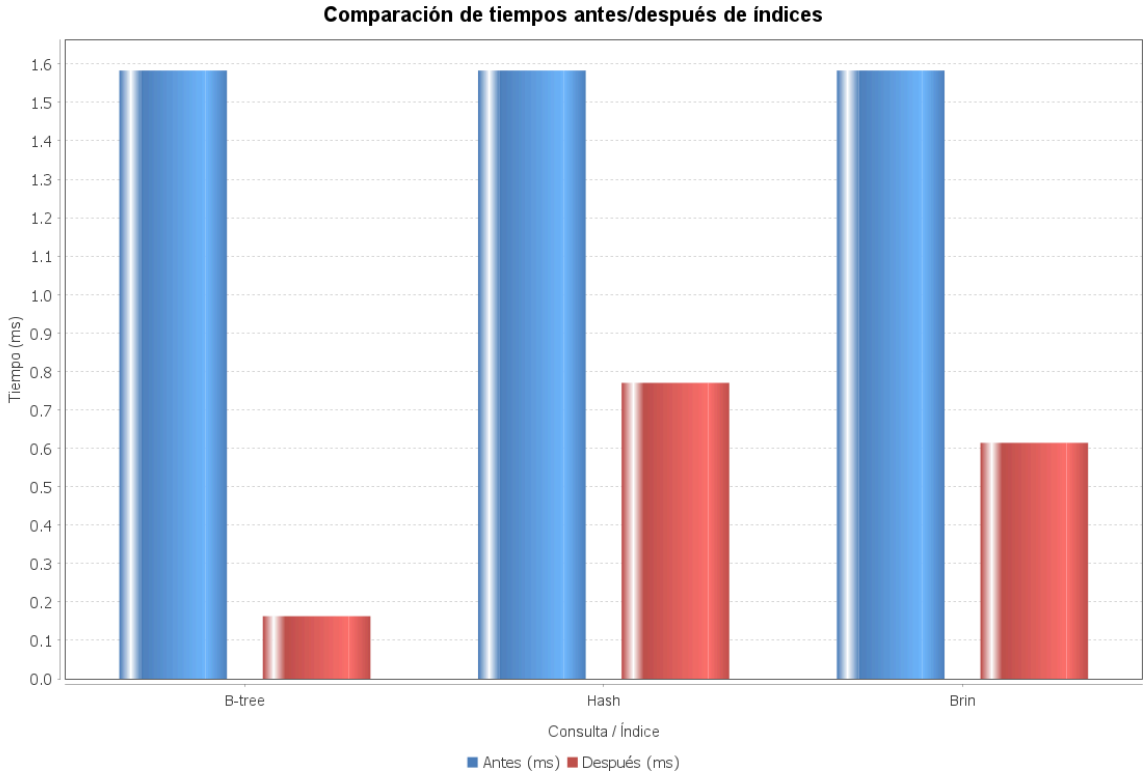


Figura #17: Comparación de los índices B-tree,Hash, BRIN en la consulta.

En la Figura #17 se puede observar el rendimiento de la misma consulta sobre tres tipos de índices diferentes en PostgreSQL: B-Tree, Hash y BRIN. En todos los casos se produce una mejora considerable en el tiempo de ejecución con respecto a la versión sin índices, pero en distinto grado según el tipo de índice. Índice B-Tree: mejor rendimiento en general, disminuyendo el tiempo de ejecución de ~1.58 ms a ~0.17 ms, siendo el más eficiente para esta consulta de igualdad y ordenamiento temporal. El índice hash también mejora el tiempo, pero en menor medida (~0.78 ms), ya que sólo optimiza búsquedas por igualdad y no por el ORDER BY. Por otro lado, el índice BRIN se desempeña en un punto medio (~0.62 ms), siendo apropiado para tablas muy grandes y datos naturalmente ordenados, pero inferior a B-Tree en este caso.

2.3. Vistas (Views)

Para la última subsección se muestran tres vistas, dos normales y una materializada. Aquí se reutilizan tres consultas ya elaboradas en la parte de queries y se transforman en vistas, ya que son consultas que se van a utilizar repetidamente para diferentes análisis. En particular, estas vistas posibilitan el cálculo y consulta inmediata de las trayectorias y distancias recorridas por cada jugador en cada partida, el historial de respuestas UX por jugador y, en el caso de la vista materializada, la distancia total y velocidad media de cada jugador en todas sus partidas, Finalmente cada vista muestra su justificación,.

Vista 1:

Para la primera vista se crea una vista para juntar en una sola estructura la distancia total recorrida y la duración (en tics) del recorrido de cada jugador en cada partida. Estos datos se recopilan y consolidan de las consultas anteriores, en las que este mismo cálculo se requería para resolver varias tareas analíticas. Al materializar estos resultados en una vista lógica, se

pueden reutilizar en futuras consultas sin tener que volver a realizar cálculos costosos con funciones de ventana, diferencia de posiciones y agrupaciones por jugador y partido.

A continuación se muestra la vista creada en PostgreSQL para la primera vista descrita anteriormente. Esta vista incluye todo el cálculo para sacar la distancia total recorrida por cada jugador en cada juego y el tiempo total en tics. Para esto usa funciones ventana (LAG()) para acceder a la posición anterior del jugador en cada tic y así calcular la distancia euclidiana entre dos posiciones consecutivas. Luego, estos desplazamientos se suman por jugador y por partida, creando así un total reutilizable.

```
CREATE VIEW doom.vw_trayectoria_partida_jugador AS

WITH eventos_ordenados AS (

  SELECT

    e.id_partida,

    e.id_jugador,

    e.tic,

    e.posicion_x,

    e.posicion_y,

    e.posicion_z,

    LAG(e.posicion_x) OVER (

      PARTITION BY e.id_partida, e.id_jugador

      ORDER BY e.tic

    ) AS pos_x_prev,

    LAG(e.posicion_y) OVER (

      PARTITION BY e.id_partida, e.id_jugador

      ORDER BY e.tic

    ) AS pos_y_prev,

    LAG(e.posicion_z) OVER (

      PARTITION BY e.id_partida, e.id_jugador

      ORDER BY e.tic

    ) AS pos_z_prev

  FROM doom.evento_telemetria e
```

```
),
trayectorias AS (
   SELECT
      id_partida,
      id_jugador,
      MIN(tic) AS tic_inicial,
      MAX(tic) AS tic_final,
      MAX(tic) - MIN(tic) AS duracion_tics,
      SUM(
         CASE
            WHEN pos_x_prev IS NULL THEN 0
            ELSE SQRT(
               POWER(posicion_x - pos_x_prev, 2) +
               POWER(posicion_y - pos_y_prev, 2) +
               POWER(posicion_z - pos_z_prev, 2)
            )
         END
      ) AS distancia_total
   FROM eventos_ordenados
   GROUP BY id_partida, id_jugador
)
SELECT *
FROM trayectorias;
```

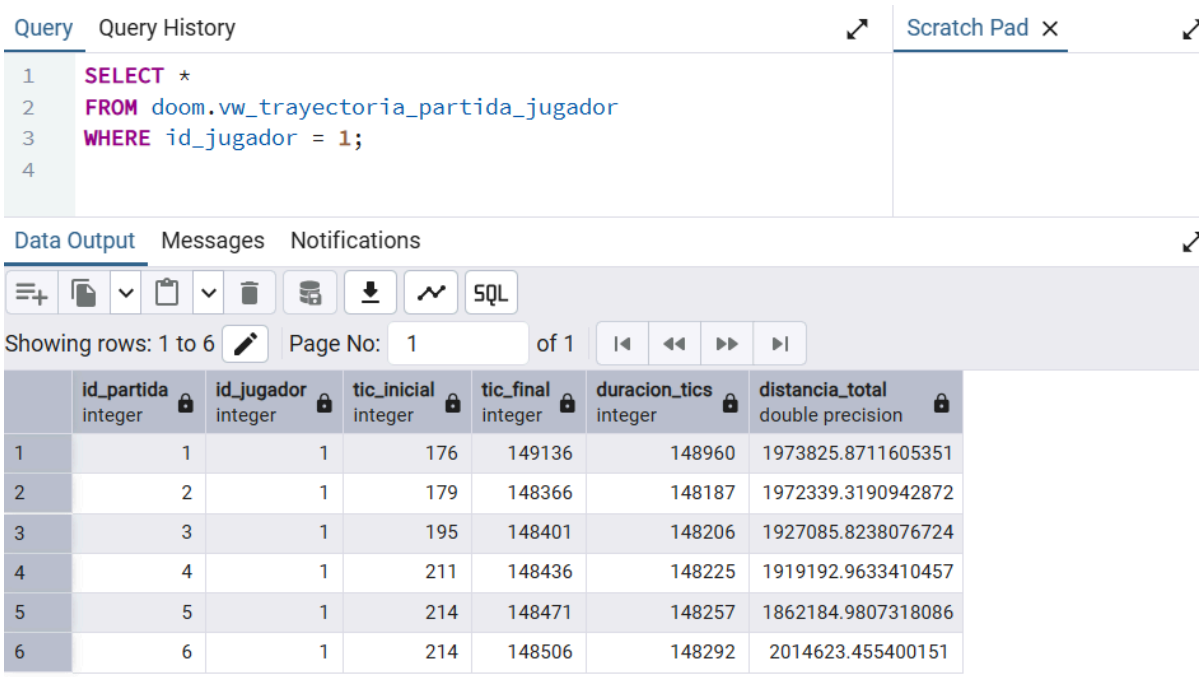
Con esto, a continuación se presenta la ejecución en PostgreSQL de la vista previamente mencionada y descrita. La vista se consultó mediante un comando sencillo que permite visualizar sus filas resultantes y verificar que las columnas generadas corresponden correctamente al cálculo de distancia total, duración en tics y demás campos definidos durante su creación.

SELECT *

FROM doom.vw_trayectoria_partida_jugador

WHERE id_jugador = 1;

En la Figura #18 se muestra el resultado obtenido de la vista creada.



	id_partida integer	id_jugador integer	tic_inicial integer	tic_final integer	duracion_tics integer	distancia_total double precision
1	1	1	176	149136	148960	1973825.8711605351
2	2	1	179	148366	148187	1972339.3190942872
3	3	1	195	148401	148206	1927085.8238076724
4	4	1	211	148436	148225	1919192.9633410457
5	5	1	214	148471	148257	1862184.9807318086
6	6	1	214	148506	148292	2014623.455400151

Figura #18: Resultado de la primera vista obtenida.

La consulta hecha sobre la vista vw_trayectoria_partida_jugador nos devuelve la información agregada para el jugador con id_jugador = 1. Los resultados muestran que la mirada está siguiendo el juego como se esperaba: para cada partida se muestran el tic inicial, el tic final, la duración en tics (su diferencia) y la distancia total recorrida, que se calcula sumando los desplazamientos euclidianos entre tics sucesivos. Los valores son consistentes: todas las partidas del jugador tienen duraciones similares; esto tiene sentido ya que el dataset está estructurado de manera uniforme.

Vista 2:

Para la segunda vista se crea una estructura que reúne toda la información de las respuestas UX de cada jugador, relacionando directamente cada respuesta con su respectivo usuario y jugador a través de los campos llave correspondientes. Esta vista combina información de las consultas utilizadas anteriormente para analizar el comportamiento y la experiencia informada por los jugadores, a la que ahora pueden accederse en conjunto los identificadores, fechas y puntuaciones de respuesta. Al encapsular esta lógica en una vista lógica, se puede reutilizar esta información en futuros análisis sin tener que volver a realizar uniones (JOIN) y filtros sobre las tablas jugador, usuario y respuesta_ux.

A continuación, la vista generada en PostgreSQL para la segunda vista descrita anteriormente: Esta vista reúne la información necesaria para analizar las preguntas UX de los jugadores, relacionando directamente la tabla respuesta_ux con la tabla jugador a través del id_usuario. La vista recoge los campos más importantes (id_jugador, alias, fecha_respuesta, puntaje_total) y los ordena para poder hacer consultas posteriores.

```
CREATE VIEW doom.vw_respuestas_ux_jugador AS

SELECT

    j.id_jugador,

    j.alias,

    u.id_usuario,

    r.id_respuesta,

    r.id_instrumento,

    r.fecha_respuesta,

    r.puntaje_total,

    r.id_pregunta

FROM doom.jugador j

JOIN doom.usuario u

    ON u.id_usuario = j.id_usuario

JOIN doom.respuesta_ux r

    ON r.id_usuario = u.id_usuario;
```

Con esto, a continuación se presenta la ejecución en PostgreSQL de la segunda vista previamente mencionada y descrita. La vista fue consultada mediante un comando sencillo que permite visualizar sus filas resultantes y verificar que las columnas generadas corresponden correctamente a la información de respuestas UX, incluyendo los datos del jugador asociado, la fecha de respuesta, el puntaje total y los demás campos definidos durante su creación.

```
SELECT *

FROM doom.vw_respuestas_ux_jugador

WHERE id_jugador = 1

ORDER BY fecha_respuesta;
```

En la Figura #19 se muestra el resultado obtenido de la vista creada.

Query | Query History | Scratch Pad

```
SELECT *
FROM doom.vw_respuestas_ux_jugador
WHERE id_jugador = 1
```

Data Output | Messages | Notifications

Showing rows: 1 to 23 | Page No: 1 of 1

	id_jugador integer	alias character varying (60)	id_usuario integer	id_respuesta integer	id_instrumento integer	fecha_respuesta timestamp without time zone
1	1	Ivancho	1	63	1	2025-11-07 19:04:33.975848
2	1	Ivancho	1	64	1	2025-11-07 19:04:54.849077
3	1	Ivancho	1	65	1	2025-11-07 19:05:41.70429
4	1	Ivancho	1	66	1	2025-11-07 19:06:28.978163
5	1	Ivancho	1	67	1	2025-11-07 19:06:34.847571
6	1	Ivancho	1	68	1	2025-11-07 19:07:02.835153
7	1	Ivancho	1	69	1	2025-11-07 19:07:08.574707
8	1	Ivancho	1	70	1	2025-11-07 19:07:13.646182
9	1	Ivancho	1	71	1	2025-11-07 19:07:18.280995
10	1	Ivancho	1	72	1	2025-11-07 19:07:24.555803
11	1	Ivancho	1	73	1	2025-11-07 19:07:38.917463
12	1	Ivancho	1	74	1	2025-11-07 19:07:43.475588

Figura #19: Resultado obtenido de la vista ejecutada.

La consulta hecha muestra de manera correcta la consolidación de toda la información de las respuestas UX del jugador con id_jugador = 1. En la tabla resultante se puede observar cómo la vista va integrando los datos de manera constante tanto de la tabla respuesta_ux como de la tabla jugador, identificando de manera explícita cada respuesta de los usuarios. Cada fila nos proporciona el alias del jugador, su id_usuario y los campos más importantes de cada respuesta: el id_respuesta, el instrumento y, sobre todo, la fecha precisa en que fue registrada la respuesta.

Vista Materializada:

Para la vista materializada se crea una estructura que almacena de forma permanente una colección de datos cuyo cálculo es más costoso o se usa en análisis posteriores. A diferencia de una vista normal, la vista materializada almacena en disco los resultados ya procesados (por ejemplo, caminos, distancias, tiempos o medidas agregadas), para que PostgreSQL no tenga que volver a calcular funciones de ventana, agregaciones o uniones costosas cada vez que se consulta. Esta estructura se apoya en una de las consultas analíticas ya elaboradas, para así conseguir la información más rápido. Con una vista materializada se mejora mucho el rendimiento en situaciones donde se requieren consultas repetitivas sobre grandes cantidades de datos.

A continuación, se muestra la vista materializada creada en PostgreSQL para la tercera vista descrita anteriormente. Esta vista guarda en disco los resultados ya calculados de una consulta analítica previamente construida, lo que permite acceder mucho más rápido a información que normalmente requiere cálculos costosos, como sumas de distancias, duración total de rutas o métricas agrupadas por jugador y por partida. La vista materializada guarda los campos tal cual se definieron en el momento de su creación y los almacena físicamente, para poder

consultarlos posteriormente sin tener que volver a calcular funciones de ventana, joins o agregaciones costosas.

```
CREATE MATERIALIZED VIEW doom.mv_resumen_distancia_velocidad_jugador AS

WITH eventos_ordenados AS (

    SELECT

        e.id_partida,

        e.id_jugador,

        e.tic,

        e.posicion_x,

        e.posicion_y,

        e.posicion_z,

        LAG(e.posicion_x) OVER (

            PARTITION BY e.id_partida, e.id_jugador

            ORDER BY e.tic

        ) AS pos_x_prev,

        LAG(e.posicion_y) OVER (

            PARTITION BY e.id_partida, e.id_jugador

            ORDER BY e.tic

        ) AS pos_y_prev,

        LAG(e.posicion_z) OVER (

            PARTITION BY e.id_partida, e.id_jugador

            ORDER BY e.tic

        ) AS pos_z_prev

    FROM doom.evento_telemetria e

),

trayectorias AS (

    SELECT

        id_partida,
```

```
        id_jugador,
        MIN(tic) AS tic_inicial,
        MAX(tic) AS tic_final,
        MAX(tic) - MIN(tic) AS duracion_tics,
        SUM(
          CASE
            WHEN pos_x_prev IS NULL THEN 0
            ELSE SQRT(
              POWER(posicion_x - pos_x_prev, 2) +
              POWER(posicion_y - pos_y_prev, 2) +
              POWER(posicion_z - pos_z_prev, 2)
            )
          END
        ) AS distancia_total
    FROM eventos_ordenados
    GROUP BY id_partida, id_jugador
),
resumen_por_jugador AS (
    SELECT
        id_jugador,
        SUM(distancia_total) AS distancia_total,
        SUM(duracion_tics)  AS duracion_total_tics
    FROM trayectorias
    GROUP BY id_jugador
)
SELECT
    j.id_jugador,
```

```
    j.alias,
    r.distancia_total,
    r.duracion_total_tics,
    CASE
        WHEN r.duracion_total_tics = 0 THEN 0
        ELSE r.distancia_total / r.duracion_total_tics
    END AS velocidad_promedio
FROM resumen_por_jugador r
JOIN doom.jugador j
    ON j.id_jugador = r.id_jugador;
```

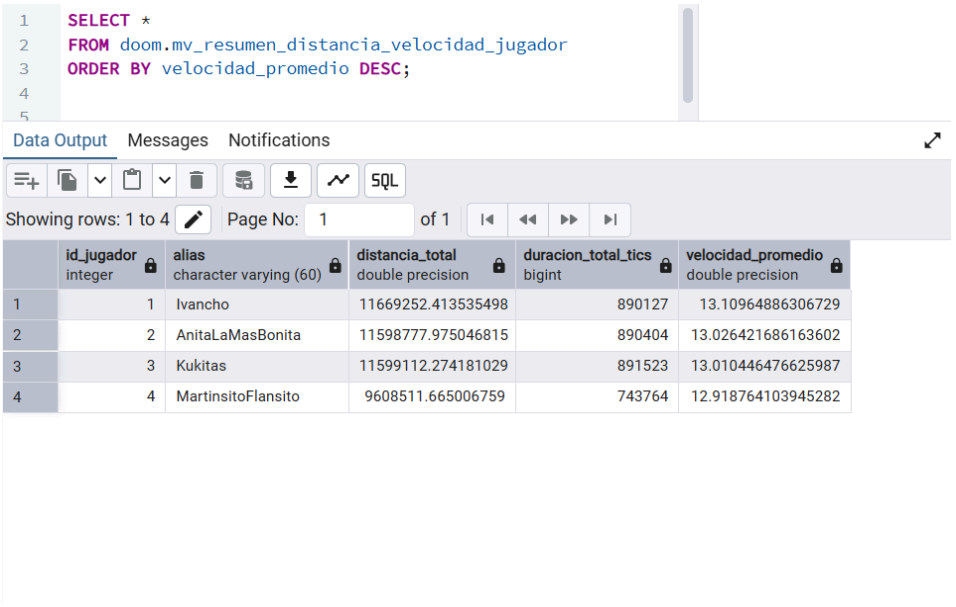
Con esto, a continuación, se muestra la ejecución en PostgreSQL de la vista materializada anteriormente descrita. La consulta se hizo con un simple comando para mostrar sus filas guardadas y verificar que los valores generados sean los mismos que la consulta original procesada. De este modo, la vista materializada siempre guarda los cálculos costosos como distancias, tiempos o métricas sumadas y estos datos pueden ser consultados directamente sin tener que repetir los cálculos.

```
SELECT *
FROM doom.mv_resumen_distancia_velocidad_jugador
ORDER BY velocidad_promedio DESC;
```

y para recargar datos se utiliza el siguiente comando

```
REFRESH MATERIALIZED VIEW doom.mv_resumen_distancia_velocidad_jugador;
```

Finalmente en la Figura #20 se muestra el resultado obtenido en la vista materializada creada.



```
SELECT *
FROM doom.mv_resumen_distancia_velocidad_jugador
ORDER BY velocidad_promedio DESC;
```

Data Output | Messages | Notifications

Showing rows: 1 to 4 | Page No: 1 of 1

	id_jugador integer	alias character varying (60)	distancia_total double precision	duracion_total_tics bigint	velocidad_promedio double precision
1	1	Ivancho	11669252.413535498	890127	13.10964886306729
2	2	AnitaLaMasBonita	11598777.975046815	890404	13.026421686163602
3	3	Kukitas	11599112.274181029	891523	13.010446476625987
4	4	MartinsitoFlansito	9608511.665006759	743764	12.918764103945282

Figura #20: Resultado obtenido de la vista materializada anteriormente descrita.

La ejecución de la vista materializada verifica que en esta se guarda de manera persistente un resumen estadístico principal del rendimiento de cada jugador en cuanto a desplazamiento dentro del juego. En el resultado se puede apreciar que para cada id_jugador se tienen 3 métricas principales: distancia total recorrida, duración total en tics y velocidad promedio (esta última obtenida dividiendo la distancia total entre la duración total). Además, se añade el nombre de jugador como alias, lo que permite entender los datos sin tener que hacer joins adicionales.

Ordenamiento por velocidad promedio: identifica rápidamente a los jugadores más rápidos en el juego. Los resultados indican distintos comportamientos entre los jugadores: Ivancho, AnitaLaMasBonita y Kukitas tienen velocidades muy parecidas, algo por encima de 13 tics por unidad de distancia, siendo MartinsitoFlansito el más lento por haber recorrido mucha menos distancia en mucho menos tiempo.

2.4. Makefile / Shell Script

El Makefile, junto con el script en bash, se encuentran documentados en el siguiente enlace a GitHub dando clic aquí. En el archivo MAKEFILE.pdf se describe paso a paso cómo levantar la base de datos, tanto de forma manual utilizando el archivo SQL de la base como mediante un shell script (.sh) que automatiza la creación del esquema y la carga de datos.

3. Conclusiones

Este proyecto valida el ciclo propuesto desde las entregas iniciales; modelado de la base de datos, la implementación física en PostgreSQL, y la carga de telemetría y posterior análisis de dichos datos. Se construyó una base de datos que almacena telemetría de juego, respuestas UX y metadatos de las partidas. Sobre esta base se implementaron consultas representativas, índices, vistas y un Makefile + script que permiten reconstruir y recrear el entorno de análisis, para comprobar su correcta implementación y funcionamiento.

El análisis de los datos brindó información valiosa sobre el comportamiento de los jugadores; sus trayectorias y velocidades medias muestran patrones similares entre sí, con excepciones por supuesto, que permiten caracterizar los distintos estilos de juego en las muestras de los participantes. Además, la vinculación entre métricas de movimiento y respuestas UX demostró que una trayectoria más corta no necesariamente implica una peor experiencia subjetiva; algunos jugadores con menos desplazamiento en el juego presentaron puntajes UX altos, cosa que sugiere que la calidad percibida del juego no depende únicamente de la movilidad del mismo.

Desde el punto de vista del rendimiento, las estrategias de indexación y materialización demostraron mejoras significativas y medibles. Los índices compuestos fueron creados para reducir el tiempo en consultas frecuentes de telemetría. La creación de vistas, particularmente de una vista materializada, permitió acelerar consultas repetitivas y simplificar la lógica analítica reutilizable. Estas optimizaciones confirman la conveniencia de incorporar decisiones de diseño físico en tareas analíticas sobre grandes volúmenes de datos, específicamente telemetría.

También se pueden reconocer limitaciones y oportunidades de mejora. La muestra de datos es relativamente homogénea y limitada en número de jugadores y episodios, lo que condiciona la generalización de los resultados. De igual forma, hay margen para enriquecer la captura de datos, como incluir más variables de telemetría, eventos contextuales, etc, e integrar índices espaciales más avanzados, como PostGIS o pipelines de ingestión incrementales para capturar datos en tiempo real. Se recomienda ampliar la colección de datos, agregar pruebas de carga más completas, automatizar métricas de benchmarking, e incluso explorar modelos de inteligencia o modelos de lenguaje predictivos que relacionen telemetría con la experiencia del usuario.

Finalmente, el proyecto demuestra que un diseño de una base de datos relacional bien pensada, acompañada de consultas analíticas robustas, y de decisiones de optimización física, permite extraer información relevante de telemetría de juego, y ligar esta información con medidas de UX. El proyecto concluye como una base reproducible y extensible lista para futuros experimentos, validaciones a mayor escala, y mejoras técnicas orientadas a análisis más complejos.

Referencias:

- Player Experience of Needs Satisfaction (PENS) – selfdeterminationtheory.org. (s. f.). https://selfdeterminationtheory.org/player-experience-of-needs-satisfaction-pens/