9.DESARROLLO EN POSTGRESQL.

9.1.Gestión de la Base de datos con PGAdmin 3.

Para una mayor comodidad de manejo de tablas y eventos, usaremos la herramienta visual PostgreSQL Administrator 3 (pgadmin III).


```
# emerge pgadmin3
```

Tras compilar e instalar, añadimos un servidor nuevo, al que especificaremos los datos de configuración de la conexión TCP/IP entre la base de datos y el Prelude-Manager.

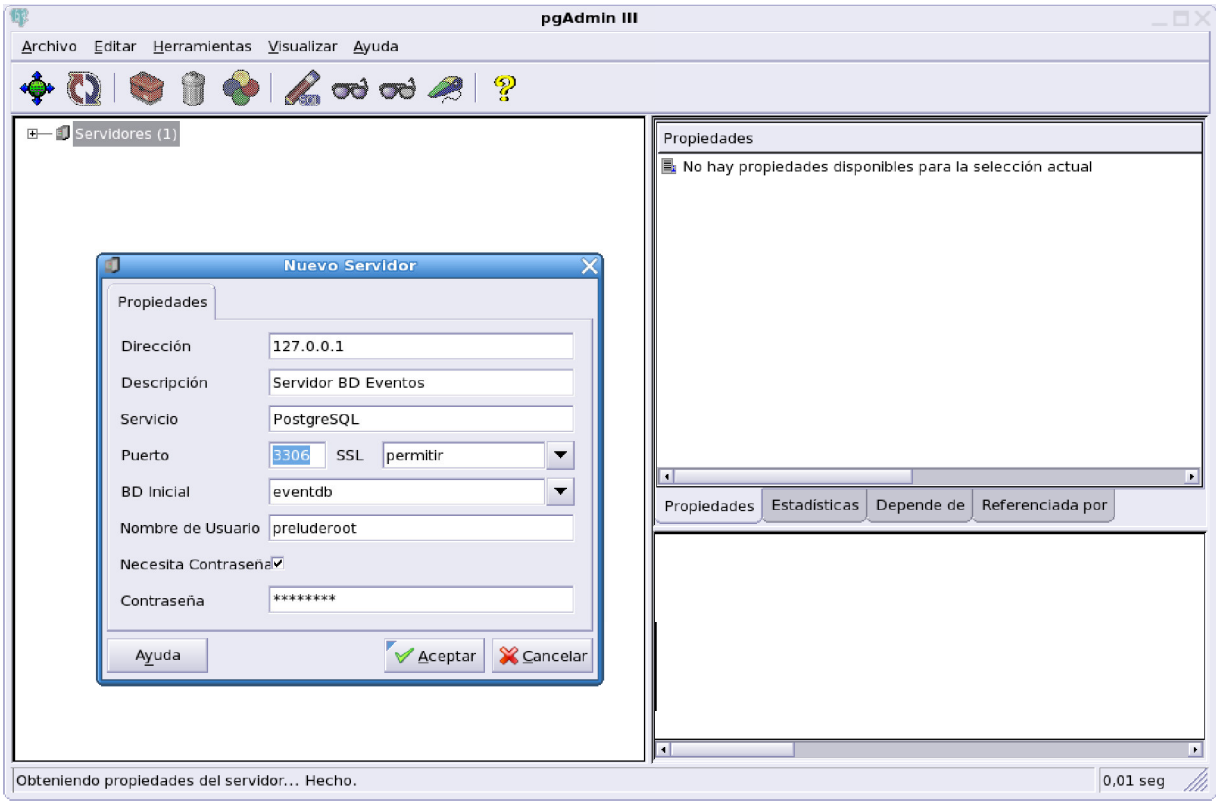


Figura 74.- Creación de un servidor de PostgreSQL con PGAdmin III.

Cuando querramos conectar a ese servidor en posteriores ocasiones, deberemos introducir la contraseña especificada en la sección [db] del archivo /etc/prelude-manager/prelude-manager.conf. Para nuestro caso el password elegido fue “proyecto”.

9.1.1.Consulta de datos.

Para ejecutar comandos SQL basta con pinchar sobre el icono en el que aparece un lápiz y las letras SQL.  En la figura 74 vemos un ejemplo para el cálculo de las sondas activas e inactivas.

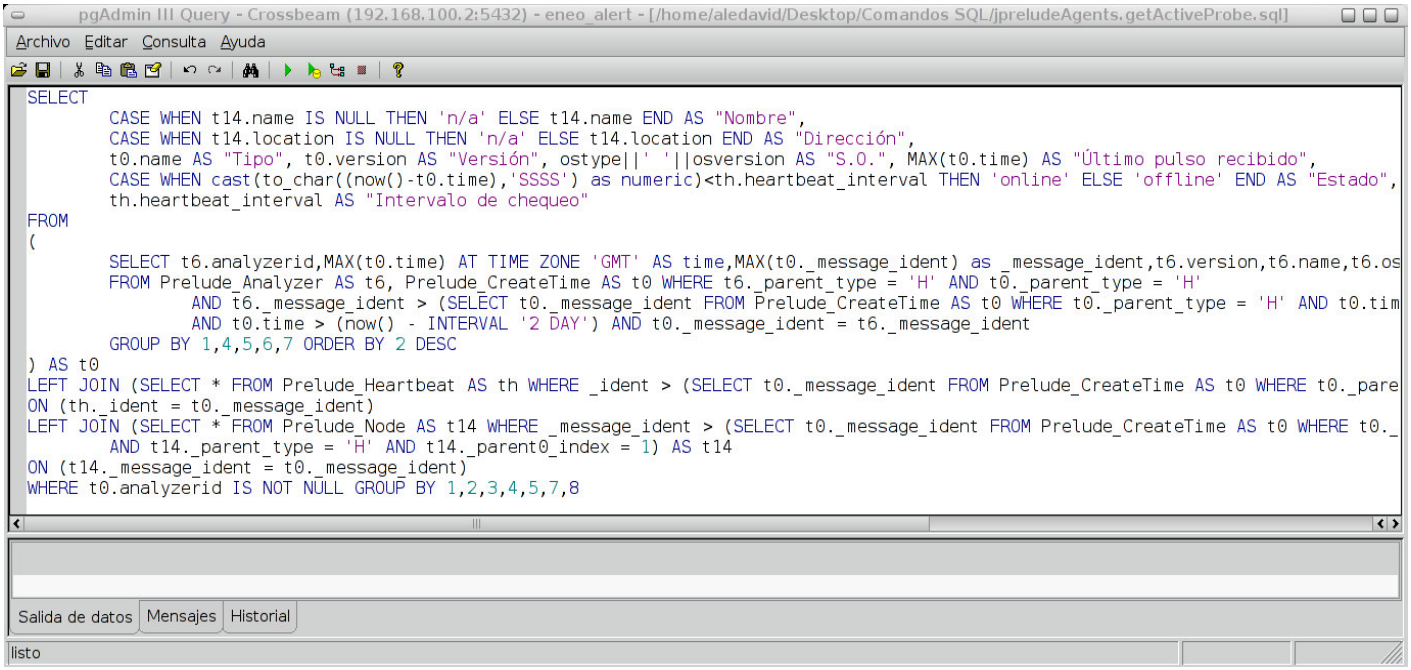
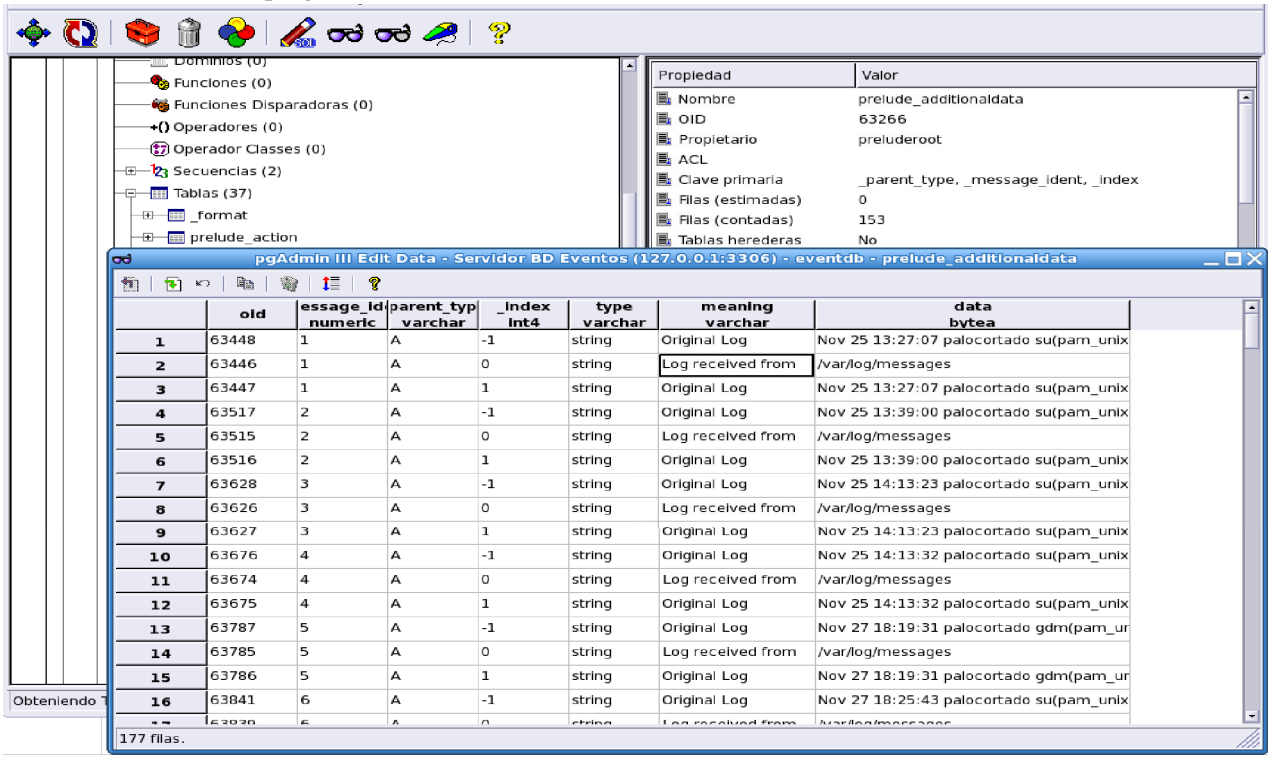


Figura 75.- Consulta de datos con PGAdmin III.

También se puede picar con el botón derecho sobre la tabla que queramos consultar y en "Ver datos" del menú desplegable.



	oid	essage_id numeric	parent_typ varchar	_index int4	type varchar	meaning varchar	data bytea
1	63448	1	A	-1	string	Original Log	Nov 25 13:27:07 palocortado su(pam_unix
2	63446	1	A	0	string	Log received from	/var/log/messages
3	63447	1	A	1	string	Original Log	Nov 25 13:27:07 palocortado su(pam_unix
4	63517	2	A	-1	string	Original Log	Nov 25 13:39:00 palocortado su(pam_unix
5	63515	2	A	0	string	Log received from	/var/log/messages
6	63516	2	A	1	string	Original Log	Nov 25 13:39:00 palocortado su(pam_unix
7	63628	3	A	-1	string	Original Log	Nov 25 14:13:23 palocortado su(pam_unix
8	63626	3	A	0	string	Log received from	/var/log/messages
9	63627	3	A	1	string	Original Log	Nov 25 14:13:23 palocortado su(pam_unix
10	63676	4	A	-1	string	Original Log	Nov 25 14:13:32 palocortado su(pam_unix
11	63674	4	A	0	string	Log received from	/var/log/messages
12	63675	4	A	1	string	Original Log	Nov 25 14:13:32 palocortado su(pam_unix
13	63787	5	A	-1	string	Original Log	Nov 27 18:19:31 palocortado gdm(pam_ur
14	63785	5	A	0	string	Log received from	/var/log/messages
15	63786	5	A	1	string	Original Log	Nov 27 18:19:31 palocortado gdm(pam_ur
16	63841	6	A	-1	string	Original Log	Nov 27 18:25:43 palocortado su(pam_unix

Figura 76.- Vista de los datos de la tabla *prelude_additionaldata*.

9.1.2.Propiedades de la tabla. Restricciones y claves.

Si queremos comprobar las claves privadas de las tablas y los campos que la forman, picamos con el botón derecho en la tabla, seleccionamos "Propiedades" y luego la pestaña de "Restricciones":

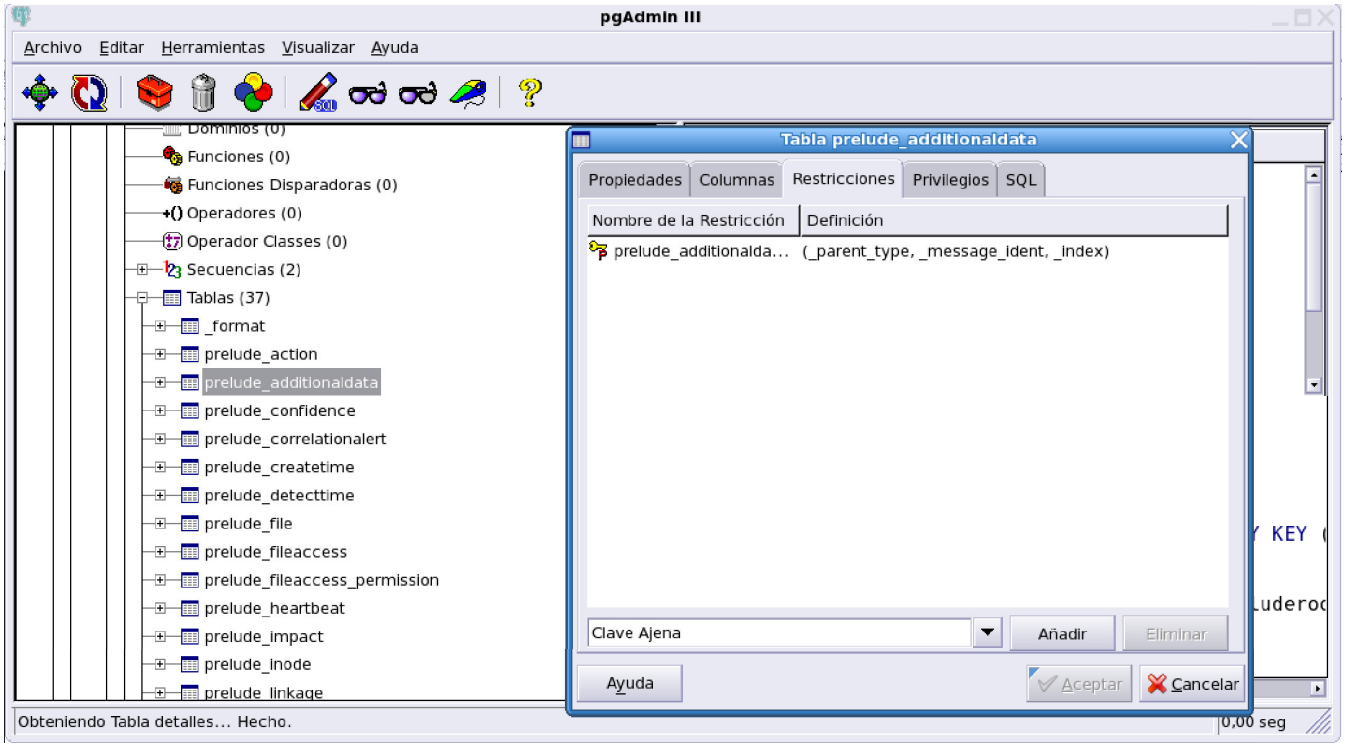


Figura 77.- Consulta de restricciones y claves con PGAdmin III.

9.2.Evolución de la estructura de las sentencias SQL.

9.2.1.Salida de depuración de Prewikka.

Dado el gran número de tablas y campos de Prelude, resulta compleja la generación de tablas similares a las de Prewikka, ya que en un principio es difícil saber de dónde obtener cada dato que queramos representar. Por tanto, primeramente se estudian los comandos que ejecuta Prewikka para conocer el nombre de las variables y tablas de las que obtiene los resultados mostrados. Posteriormente, en el siguiente punto, se explica brevemente la mejora respecto de Prewikka y los resultados empíricos de una prueba de carga.

Para ver los comandos que ejecuta Prewikka, activamos el nivel de depuración de PostgreSQL **número 2** ya que el primero no muestra los comandos SQL y el 5 muestra demasiada información y es difícil leer las peticiones de la salida de depuración entre tantos datos. Para ello activamos la opción **-d 2**.

```
[postgres@palocortado]$ postmaster -D /var/lib/postgresql/data -d 2 -p 5432
```

Los datos que se muestran en la consola se copian y pegan en un fichero de texto al que se el pasa un archivo que lo procesa.

El guión utilizado para realizar esto es prewikka2pgadminsql.sh, haciendo uso de los ficheros de logs de postgresql, o bien también puede copiarse la salida de depuración de postmaster a un fichero que sea atacado por el *awk*:

prewikka2pgadminsql.sh

```
awk -f /var/lib/postgresql/prewikka-sql/prewikka2pgadminsql.awk
     /var/lib/postgresql/data/pg_log/postgre*.log >> /var/lib/postgresql/query.sql
```

prewikka2pgadminsql.awk

```
BEGIN {
   msg_ok="LOG:";
   dustbin="DEBUG:";
   print("#Fichero de comandos sql.");
   nline=0;
   line="";
}

{
   #Procesamiento por lotes del fichero prewikka.sql
   if ((msg_ok == $1) && ("sentencia:" == $2))
   {
      nline++;
      FS="";
      $1 = "";
      $2 = "";
      FS=" ";
      for (n=3;n<=NF;n++)
         line+="$n ";
      printf("//%d.-\n",nline)
      print($line);
      line="";
   }
}
```

El fichero devuelto tiene este formato:

query.sql

```
#Fichero de comandos sql.
//1.-
  SELECT * FROM Prewikka_Version
[...]
//14.- Calcula la tabla de el numero de alertas (agrupadas por direcciono origen y
       destino) y entra en un bucle con la primera fila.
  SELECT t0.address, t1.address, COUNT(t2.time), MAX(t2.time)
  FROM Prelude_Alert AS top_table
  LEFT JOIN Prelude_Address AS t0
  ON (t0._parent_type='S' AND t0._message_ident=top_table._ident
    AND t0._parent0_index = 0 AND t0._index = 0)
  LEFT JOIN Prelude_Address AS t1 ON (t1._parent_type='T'
    AND t1._message_ident=top_table._ident AND t1._parent0_index = 0 AND
    t1._index = 0)
  LEFT JOIN Prelude_CreateTime AS t2
  ON (t2._parent_type='A' AND t2._message_ident=top_table._ident)
  WHERE t2.time >= '2005-11-30 23:00:00' AND t2.time < '2005-12-20 23:00:00'
  GROUP BY 1, 2 ORDER BY 4 DESC
```

9.2.2.Optimización del tiempo de búsqueda.

Las consultas de Prewikka se basan en hallar una lista de identificadores de mensaje para la que se ejecuta un bucle que obtiene cada campo para cada mensaje. Si a este ineficiente método sumamos lo lento de una interfaz web respecto de una en Java, concluimos que es mejor realizar consultas de bloque indexadas, y no consultas para cada identificador de alerta.

9.2.2.1. Mejora 1: Reducir el número de consultas usando índices.

Por tanto, las optimizaciones se basan en reducir el número de consultas. Excepto el primer nivel *martealertFirstLevel*, **todas** las tablas se obtienen con una única consulta de sentencias SELECT encadenadas. Se comienza con una SELECT, seguido de las variables, seguido de la cláusula FROM y a continuación las tablas que nunca poseerán valores nulos y por tanto tendrán entrada para todas las alertas. Se trata de la variable *text* para *prelude_classification* y de las variables *severity, completion* para la tabla *prelude_impact*.

9.2.2.2. Mejora 2: Uso de LEFT JOIN en lugar de RIGHT JOIN o INNER JOIN.

Luego se hacen operaciones de JOIN con cada tabla cuyos datos también queramos obtener. Si queremos representar por ejemplo la dirección IP origen, basta con hacer un LEFT JOIN con la tabla *prelude_address*. El JOIN a izquierdas o LEFT JOIN garantiza mayor velocidad que el RIGHT JOIN para esta estructura de comandos a la vez que conserva el número de filas de la tabla a izquierdas, en caso de que tenga más filas que la de la derecha. Esto se da cuando hacemos un LEFT JOIN de la tabla *prelude_alert* (que posee entradas para todas las alertas) con la de *prelude_address* ([31]) que no posee entradas cuando la IP origen es la 127.0.0.1 (localhost).

Ejemplo: Si tenemos la tabla *prelude_address* con cinco filas para IP origen y en la *prelude_alert* tenemos 10 filas, se concluye que tenemos 5 direcciones IP origen que son localhost y otras 5 que no lo son. Si hiciéramos otro JOIN que no fuera a izquierdas, o bien produce los resultados más lentos que el LEFT JOIN o bien se obtendrían sólo 5 filas ya que al hacer *WHERE prelude_address._message_ident = prelude_alert._ident* se pierden las filas cuyo identificador no existe en la tabla de direcciones. Con el LEFT JOIN aparecen las 10 filas y las que no existen en *prelude_address* se muestran con campo con valor NULL. Éste campo a NULL se soluciona fácilmente con una modificación de la variable del SELECT: SELECT CASE WHEN Prelude_Address.address IS NULL THEN '127.0.0.1' ELSE Prelude_Address.address END, ...

9.2.2.3. Mejora 3: Cribado de las tablas por filas y columnas: optimización del producto cartesiano intrínseco a los JOIN.

Además, las tablas que efectúan los JOIN están filtradas por filas y columnas. Por filas usan el filtro *martealertHashFilter* y por columnas sólo devolviendo al SELECT del nivel superior las columnas cuyos datos queremos representar. Con esto disminuimos el número de filas de los productos cartesianos que llevan implícitos todos los tipos de JOIN.

9.2.2.4. Mejora 4: Uso de GROUP BY con SELECT en lugar de SELECT DISTINCT.

Por último, se comprueba empíricamente que el uso de la cláusula GROUP BY con sentencias SELECT anidadas es más efectivo que el usar SELECT DISTINCT anidadas.

9.2.2.5. Mejora 5: No usar operador "!=" en la cláusula WHERE.

Es más eficiente el realizar: *consulta EXCEPT consulta WHERE id IS NULL* que obtener los mismos datos ejecutando: *consulta WHERE id IS NOT NULL*.

[31] Para obtener la IP origen es necesario filtrar en el WHERE por el índice *_parent_type = 'S'*.

9.2.2.6. Evitar el uso de operadores lógicos OR combinados sin paréntesis.

Debe de evitarse el uso de filtros del tipo *WHERE a = 1 OR b = 2 AND c = 3 ...* sobre todo cuando el filtro puede llegar a ser extenso. Esto produce una elevada cantidad de combinaciones debido a la propiedad distributiva de las operaciones lógicas.

9.2.2.7. Conclusión y comparativa con tiempos de Prewikka.

De todo esto y para cada tabla bien sea *martealertFirstLevel, martealertTopStats, martealertAgents* y *martealertRealTime* se han almacenado comandos en ficheros ".sql" con el nombre de la tabla representada y un seguimiento de versión de los comandos.

Resumiendo todos los conceptos anteriores, se consigue mejorar Prewikka obteniendo los mismos datos en un tiempo más de 80 veces inferior que al de Prewikka. Para pocas alertas la diferencia no es perceptible pero para un número de alertas de un orden de magnitud 1000 Prewikka tarda del orden del minuto y *martealert* del orden de pocos segundos.

Si aumentamos el número de alertas a centenares de miles de alertas, *Marte Alert* consigue devolver los resultados de los tops en decenas de segundos mientras que Prewikka se cuelga y no es capaz de operar con tantos datos.

9.3.Comandos SQL.

Para comprender la estructura de los comandos SQL es necesario ver un ejemplo de uno de ellos. No obstante, dada la complejidad de dicho comando, es recomendable en primer lugar analizar el código del método *getCommand* usado en las subclases de *martealertTable*.

9.3.1.Método *getCommand*.

A continuación se detalla el código del método que genera los comandos SQL desde la clase *martealertTopStats*.

```
public String getCommand(martealertHashFilter filter) {
        int total = filter.nfilteredAlerts;
[...]
        String subSql = "SELECT "+getSqlVars(total)+
            "FROM "+martealertSqlUtils.getSqlCommand(filter)+" ";
        String subWhere = "GROUP BY "+getSqlGroupedBy();

        return  "SELECT * FROM ("+subSql+subWhere+") AS t " +
                "EXCEPT " +
                "SELECT * FROM ("+subSql+" WHERE tlast._ident IS NULL "+
                     subWhere+") AS t " +
                "ORDER BY "+getSqlOrderBy()+this.limit;
```

En este código se invocan los métodos que conforman el comando SQL, el cual se compone por el mismo comando ***subSql*** excepto él mismo cuando el índice es NULL. Esto es mucho más eficiente que no usar el EXCEPT y filtrar por WHERE tlast._ident IS NOT NULL, tal y como se explica en la Mejora nº5.

El método `martealertSqlUtils.getSqlCommand(filter)` compone el comando según los valores que `getSqlVars` necesite y que el filtro indique. Para ello se usa el objeto *jpFields* del *martealertHashFilter*. A continuación un comando de ejemplo que calcula el segundo nivel: (filtrando por pareja de IP origen y destino)

Comando para la generación de *martealertSecondLevel*:

```
SELECT COUNT(tlast._ident)||' x '||text END AS "N x (Tipo de mensaje)" ,
    severity AS "Gravedad" ,
    CASE WHEN (completion = 'succeeded' OR completion IS NULL) THEN TRUE
        ELSE FALSE END AS "Completado" ,
    CASE WHEN (t2.s_addr) IS NULL OR t2.s_addr = ' ' THEN '127.0.0.1'
        ELSE t2.s_addr END AS "IP Origen" ,
    CASE WHEN (t_addr) IS NULL THEN '127.0.0.1' ELSE t_addr END AS "IP Destino" ,
    CASE WHEN iana_protocol_number IS NULL THEN '0'
        ELSE iana_protocol_number END AS "Protocolo" ,
    CASE WHEN MIN(s_port) IS NULL THEN '0' ELSE MIN(s_port) END AS "Puerto Orig",
    CASE WHEN port IS NULL THEN '0' ELSE port END AS "Puerto Destino" ,
    name AS "Nombre" ,
    CASE WHEN a_addr IS NULL THEN '127.0.0.1' ELSE a_addr END AS "Sonda" ,
    MIN(t0.time) AS "Desde" ,
    MAX(t0.time) AS "Hasta"
```

Hasta aquí las variables, las cuales incluyen la representación de los valores por defecto si se encuentra NULL. Lo siguiente son las tablas cuyos valores nunca son NULL. Como tabla top se usa una consulta SELECT cribada entre la fecha indicada, devolviendo los identificadores para hacer los JOIN. Ésta tabla se detalla en negrita y se nombra como t0.

```
FROM
(
    SELECT t0._message_ident,t0.time AT TIME ZONE 'GMT' AS time, t1.text,
        t7.severity, t7.completion
     FROM
        (SELECT t0._message_ident,t0.time FROM
           (SELECT t0._message_ident,t0.time,t0._parent_type FROM
           Prelude_CreateTime as t0 WHERE t0._parent_type = 'A' AND
           (t0.time >= '2006-03-21 12:50:06' AND t0.time <=
           '2006-03-21 13:30:52')
        ) AS t0)  AS t0, Prelude_Classification  AS t1 , Prelude_Impact  AS t7
     WHERE t0._parent_type = 'A' AND (t0.time >= '2006-03-21 12:50:06' AND
         t0.time <= '2006-03-21 13:30:52')
    AND  t1._message_ident =t0._message_ident
    AND  t7._message_ident =t0._message_ident
    GROUP BY 1,2, 3, 4, 5, 6, 7, 8
) AS t0
```

Además se ha devuelto la variable t0.time en hora de GMT, para unificar la referencia independiente de la hora del sistema local. A continuación, el JOIN de la tabla anterior con las de los parámetros que puedan ser NULL.

```
LEFT JOIN
(SELECT t2._message_ident ,t2.address as s_addr FROM
    (SELECT t2._message_ident ,t2.address, t0.time FROM Prelude_CreateTime  AS t0,
    Prelude_Address  AS t2 WHERE ((t2.address = '192.168.100.60'))
    AND (t0.time >= '2006-03-21 12:50:06' AND t0.time <= '2006-03-21 13:30:52')
    AND t0._parent_type = 'A' AND t2._index = 0 AND t2._parent_type = 'S'
    AND t2._message_ident  = t0._message_ident  GROUP BY 1,2,3 ) AS t2
)  AS t2
```

```
ON (t2.s_addr = '192.168.100.60' AND t0._message_ident =t2._message_ident )

LEFT JOIN
(SELECT t3._message_ident ,t3.address as t_addr FROM
    (SELECT t3._message_ident ,t3.address, t0.time FROM Prelude_CreateTime  AS t0,
    Prelude_Address  AS t3 WHERE (t0.time >= '2006-03-21 12:50:06'
    AND t0.time <= '2006-03-21 13:30:52') AND t0._parent_type = 'A'
    AND t3._parent_type='T' AND t3._message_ident = t0._message_ident
    AND t3._index = 0 GROUP BY 1,2,3 ) AS t3
) AS t3
ON (t3.address = '127.0.0.1' AND t0._message_ident =t3._message_ident )

LEFT JOIN
(SELECT t11._message_ident ,t11.port AS s_port FROM
    (SELECT t11._message_ident ,t11.port, t0.time FROM Prelude_CreateTime  AS t0,
    Prelude_Service  AS t11 WHERE (t0.time >= '2006-03-21 12:50:06'
    AND t0.time <= '2006-03-21 13:30:52') AND t0._parent_type = 'A'
    AND t11._parent_type='S' AND t11._message_ident  = t0._message_ident
    GROUP BY 1,2,3 ) AS t11
) AS t11
ON (t0._message_ident =t11._message_ident )
[...]
```

Este JOIN para obtener el puerto origen tiene la misma estructura que los siguientes para calcular los datos: puerto destino, sensor y sonda. Por tanto pasamos directamente al último JOIN que es especial porque filtra por el identificador de mensaje NULL. Esto se hace de esta forma porque conforme se van encadenando los LEFT JOIN en serie, en caso de filtrar, las alertas que no encajen con el filtro pasan a tener identificador NULL. Basta con remarcar y unificar los filtros de todos los campos en el último JOIN y hacer tlast._ident que no sea NULL para obtener los datos deseados.

Además, el filtro del último JOIN puede incluir filtrado del tipo **`s_addr IS NULL`**, cosa que no se puede incluir en el LEFT JOIN de IP origen, ya que incluiría las alertas descartadas por el filtro para los JOIN anteriores al de IP origen.

```
[...]
LEFT JOIN
(SELECT _message_ident AS _ident,time FROM
    (SELECT t0._message_ident,t0.time FROM Prelude_CreateTime AS t0
    WHERE t0.time >= '2006-03-21 12:50:06' AND t0.time <= '2006-03-21 13:30:52'
    AND _parent_type = 'A' GROUP BY 1,2) AS t0
) AS tlast
ON (s_addr = '192.168.100.60' AND t_addr = '127.0.0.1'
    AND t0._message_ident =tlast._ident )

WHERE tlast._ident IS NOT NULL GROUP BY t0.text,2,3,4,5,6,8,9,10
ORDER BY 7 DESC,3,4
```

Por simplicidad no se ha detallado el comando entero ya que la estructura resulta repetitiva y con filtrado por distintos campos puede llegar a rebasar los 4000 caracteres de longitud.

9.4.Reindexado de libpreludedb.

La librería de LibpreludeDB está indexada de manera adecuada para la interfaz de Prelude Prewikka. No obstante, se puede comprobar con pruebas de carga si se define un índice con tres campos como por ejemplo *prelude_analyzer_index_model(_parent_type,_index,model)*, al realizar una consulta en la que se filtre por dos de ellos([32]), la consulta puede ir de 6 a 10 veces más lenta en media. Entonces hay dos opciones; bien crear otro índice que tenga sólo dos campos, o bien uno con tres campos en el que el tercero sea de utilidad para el filtrado. En el código SQL siguiente se ha comentado la opción de los dos campos.

```
DROP INDEX prelude_analyzer_index_model ON
                                Prelude_Analyzer (_parent_type,_index,model);
-- CREATE INDEX prelude_analyzer_index ON Prelude_Analyzer (_parent_type, _index)
CREATE INDEX prelude_analyzer_index_name ON
                                Prelude_Analyzer (_parent_type,_index,name);
REINDEX TABLE Prelude_Analyzer;
```

Se ha optado por cambiar *model* por *name* debido a que *model* puede ser NULL. Para tener esto en cuenta hay que añadir cláusulas “CASE WHEN” en las variables del SELECT, lo que ralentiza las consultas.

Por último, se reindexa la tabla *Prelude_Analyzer*.

[32] Por ejemplo: SELECT * FROM prelude_analyzer WHERE _parent_type = 'S' AND _index = 0 irá mucho más lenta que SELECT * FROM prelude_analyzer WHERE _parent_type = 'S' AND _index = 0 AND model IS NULL.