Aplicación de PVM a herramientas de ETL

Abstract

En este documento se presenta un trabajo realizado con la herramienta de programación paralela PVM sobre el área de procesos de extracción transformación y carga (ETL). Se muestra una herramienta con la cual se permite modelar grafos basados en la idea de "*flujo de datos*" para la resolución de problemas en el área de ETL y ejecutarlos en forma centralizada o distribuida. Se presenta también un problema de ejemplo en el área de generación de redundancia para el análisis OLAP y su resolución con la herramienta.

Introducción

El objetivo del trabajo es formar experiencias en el trabajo con PVM para esto se implementara una aplicación que utilice esta tecnología. El área sobre la que se trabajara será modelar procesos de ETL utilizando grafos, estos grafos podrán ser interpretados por la herramienta propuesta de forma de poder ejecutarlos distribuidamente. En esto ultimo será donde se vera su intersección con la herramienta PVM.

Además se tratara de interceptar este trabajo de forma que pueda ser interesante para los objetivos del grupo de Concepción de Sistemas de Información, en esta línea se planteara una posible implementación afín de la primitiva "**P9 Aggregate Generation**" presentada en [AM2000].

Finalmente esta implementación se aplicara en un ejemplo al problema de generación de redundancia masiva para análisis OLAP. Para esto se modelara un grafo que representara una forma de generar la redundancia, esto claramente puede ser pensado como un proceso de ETL. El hecho de ser un problema de un alto potencial de crecimiento en por ejemplo dos líneas: cantidad de datos para procesar y dificultad inherente por la cantidad de dimensiones y jerarquías involucradas es lo que lo hace interesante como ejemplo.

En lo que sigue primero se presentaran las áreas de trabajo relacionadas, luego se pasara directamente a la herramienta implementada deteniéndose en la implementación distribuida (en la sección "Implementación Distribuida") y en la implementación de la primitiva P9 (en la sección "Implementación del AGREGATE OPERATOR (P9)"). A continuación se presentara el ejemplo, se comentaran las posibles líneas de trabajo futuro y se finalizara con las conclusiones.
Se incluyen dos apéndices con la implementación del ejemplo expuesto en forma completa y con los comandos disponibles en la herramienta.

Trabajos Relacionados

La implementación de la herramienta propuesta se realizo en base a una modificación al trabajo presentado en [BBGS] y [MG2000] para adaptarla al contexto y alcance del presente trabajo. Sobre la temática presentada en estos dos artículos se realizo un análisis previo al comienzo del proyecto que se presenta en [IL2001].

A modo de resumen [BBGS] y [MG2000] podemos decir que lo que ahí se presenta es un prototipo para procesos paralelos de ETL, basado en el encare de *flujo de datos* (el cual señala ha sido tomado por la comunidad de base de datos), montado sobre un cluster de computadores.

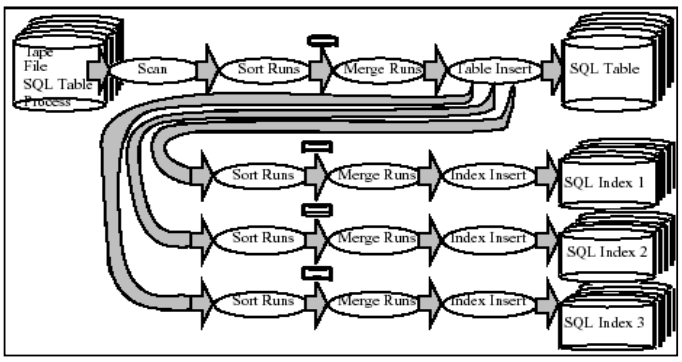

Figura 1: Ejemplo de grafo de definición.

En el encare basado en el *flujo de datos*, la información reside en

archivos o bases de datos en algunos discos, cintas u otros dispositivos de memoria. Los algoritmos buscan subconjuntos de esta información, y o bien depositan sus resultados en dispositivos de almacenamiento o retornan sus respuestas a un array de programas de aplicación. Un algoritmo de flujo de datos se describe como un grafo dirigido.
Los nodos del grafo, llamados operadores, son programas secuenciales. Cada operador lee sus *"streams"* de entrada (*"dataflows"*), transforma la información, y produce uno o mas *"streams"* secuenciales de salidas de datos. Las aristas del grafo muestran los flujos de datos entre operadores. En la Figura 1 se presenta un ejemplo de un proceso modelado con esta técnica.

Sobre la herramienta PVM, lo que podemos comentar es que se trata de una solución para atacar problemas de programación paralela. Es una framework es decir, da soporte para resolver este tipo de problemas, no los resuelve[1].
PVM compuesto por una "*maquina virtual*" (MV), que para expresarlo en pocas palabras es una herramienta de control distribuido de procesos, similar a lo que seria una "scheduler" del sistema operativo. Y por otro lado las bibliotecas de programación necesarias para interactuar con la MV y los demás procesos que nos permiten comunicarlos dentro de la arquitectura propuesta.
Esta MV se conforma por procesos gemelos que corren en todos los nodos de la red que a su vez la definen. Brinda servicios a los procesos que trabajan bajo su ambiente y se ejecuta en todos los nodos (maquinas) que la conforman, de ahí su nombre.

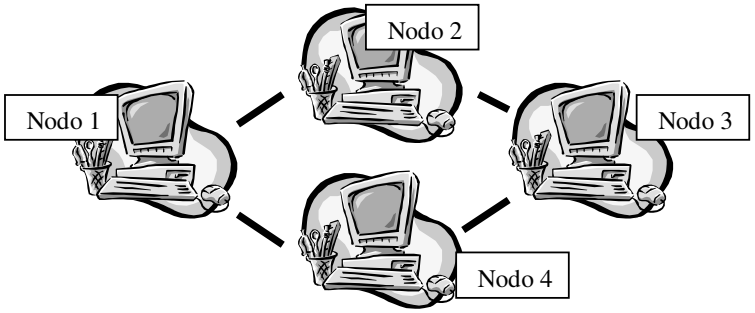

Figura 2: Nodos en una maquina virtual PVM.

Para el control de esta infraestructura se tiene una consola desde la cual se pueden monitorear los procesos que ejecutan dentro de la maquina virtual, así como alterar la configuración, por ejemplo agregando o removiendo una maquina, etc.

La herramienta

La herramienta que se implemento ataca tres problemas, que son, la definición del proceso de extracción, transformación y carga (ETL)[2], la ejecución centralizada y la ejecución distribuida.

Para enfrentar el primer problema se da acceso a una serie de comandos que permiten definir los distintos elementos del proceso. Esto según se extrae del documento serian, la definición del grafo del proceso con los tipos de operaciones que cada nodo ejecuta y los formatos de los registros intercambiados en cada arista[3].

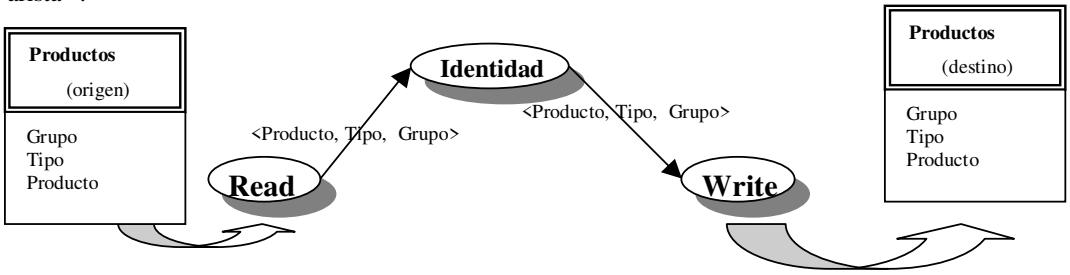

Figura 3: Ejemplo de proceso.

[1] Por mas información sobre PVM consultar [PVM94].
[2] Obviamente aquí se esta abusando de la palabra ETL ya que lo que se encuentra realizado simplemente ataca un sector acotado y reducido de los problemas en el área.
[3] Estamos tomando una solución simplificada de la propuesta original.

Supongamos el siguiente ejemplo para fijar ideas. Sea un conjunto de productos identificados por un código teniendo a su vez un tipo de producto asociado y un grupo.
La tabla relacional origen de los datos seria entonces de tuplas formadas por <Producto, Tipo, Grupo>. Supongamos a su vez la transformación mas simple posible, la identidad. Y finalmente dado que estamos aplicando la identidad la tabla relacional destino será del mismo formato que la tabla origen. Gráficamente esto se ve en la Figura 3.

Para darle la información a la herramienta disponemos entre otros de los siguiente comandos:

- "***define record format***". Que nos permite definir los formatos de registro que se usaran para intercambiar la información entre los distintos nodos del sistema (Producto, Tipo, Grupo), así como leer y almacenar la información de archivos. Estos formatos son dados por un nombre y un conjunto de triplas formadas por "*nombre"*, "*tamaño"* y "*tipo"* de cada atributo (p.e. "Formato1" = {<Producto, 10, String>, <Tipo, 4, Entero>, <Grupo, 4, String>}).

  En sintaxis de la herramienta[4, 5]:
  define record format Formato1 { Producto, 10, String; Tipo, 4, Integer; Grupo, 4, String};

- "***define nodes***". Que nos permitirá definir los nodos que habrá involucrados en el grafo que define el proceso (Read, Identidad y Write). Estos serán dados por conjuntos de duplas "*nombre"* y "*operador"* (p.e. {<"Read", Read>, <"Identidad", Identidad>, <"Write", Write>}). Los operadores son los que nos permitirán definir la operación a realizar en cada nodo (p.e. el nodo de nombre "Identidad" realiza la operación de identidad, es decir copia toda su entrada en la salida).

  En sintaxis de la herramienta[5]:
  define nodes {
  NodoRead, read operator (W:\cprojects\data\ventas-origen.txt);
  NodoIdentity, identity operator;
  NodoWrite, write operator (W:\cprojects\data\ventas-destino.txt)
  };

  Los operadores que tenemos disponibles para trabajar y por lo tanto la funcionalidad que podremos colocar en cada nodo es:

  - **READ OPERATOR**. Lo que nos permitirá es leer la información de un archivo dado y colocarla en las aristas de salida para que otros nodos la utilicen. Como se ve recibe como parámetro el nombre del archivo del cual leerá los datos.

  - **WRITE OPERATOR**. Realiza la operación inversa a la anterior, toma su entrada de las aristas que le llegan y la vuelca en el archivo que se le indica. También recibe como parámetro el nombre del archivo del cual grabara los datos.
    Aunque uno intuitivamente pensaría que tanto en este como en el operador anterior es necesario recibir el formato de los datos que se leerán o escribirán por una decisión de diseño este se da con las aristas que salen del nodo, esto simplemente porque parece mas flexible que el otro enfoque.

  - **IDENTITY OPERATOR**. Copia la información de todas las aristas de entrada en las aristas de salida y obviamente no recibe parámetros.

  - **AGGREGATE OPERATOR**. Toma todas sus entradas, y realiza la agregación de estos colocándolos luego en todas sus salidas. La inspiración de este operador es claramente la primitiva "**P9 Aggregate Generation**" presentada en [AM2000] y recomendamos referirse a esta referencia por mas información. De igual forma a continuación nos extenderemos un poco mas sobre la resolución en particular.

[4] Hemos de notar que aunque se hablo del grafo primero y de la definición de formatos posteriormente, esto en la practica se realiza en forma inversa ya que como veremos las definiciones de formato son necesarias para la definición de las aristas.
[5] Por mas información ver en el "Apéndice – Comandos de la Herramienta" el comando **define**.

- "***define edges***". Que definirá las aristas que unen estos nodos para formar el grafo. Estas aristas están dadas por un conjunto de cuádruplas "*nombre*", "*nodo origen*", "*nodo destino*" y "*formato*" (p.e. {<"AristaRead–Identidad", Read, Identidad, "Formato 1">...}).

  En sintaxis de la herramienta[5]:
  define edges {
  AristaRead-Identidad, NodeRead, NodeIdentity, Formato1;
  AristaIdentidad-Write, NodeIdentity, NodeWrite, Formato1
  };

Con estos comandos y algunos detalles mas especificamos el proceso que queremos realizar[6], pasemos ahora a como lo ejecutamos. Aquí como ya mencionamos tenemos dos posibilidades: ejecución centralizada y ejecución distribuida.
Que queremos decir con ejecución centralizada?. Bueno nos referimos a que un solo proceso del sistema operativo realizara toda la tarea y obviamente en una sola maquina. Esto fuera de detalles de implementación no es mas que un algoritmo que hace una recorrida del grafo y va calculando los resultados.
Dada la implementación actual la única restricción que impondremos sobre el grafo (en el caso centralizado) es que este sea un árbol (o bosque), dado que la recorrida se realiza con invocaciones recursivas y estas no están preparadas para tomar entrada de varias fuentes. Por las dudas remarcamos que en el contexto distribuido esto no es así.

Para llevar a cabo la ejecución centralizada basta simplemente que una vez definido lo anterior ejecutemos el comando "**execute alone**".

Implementación Distribuida

Bueno y ahora el punto mas interesante, que es lo que sucede con la ejecución distribuida?. Aquí entraremos ahora si en un poco mas de detalle sobre como se implemento.
Primero que nada lo que ya indicamos sobre la definición del proceso es exactamente igual y a los efectos de la persona que lo ejecuta solo basta con utilizar ahora el comando "**execute**" en vez del otro.

La diferencia es que ahora en vez de utilizarse el proceso actual para realizar el procesamiento, se levantaran N procesos (N es la cantidad de nodos del sistema) representando los nodos del sistema. Luego de estar estos en ejecución mediante las facilidades de PVM se le enviaran los datos sobre la estructura del grafo que les interesa (básicamente el sub-grafo que lo tiene a el y a los nodos que se encuentran a un salto de distancia, además de las aristas de salto). Desde ese momento en adelante cada nodo simplemente empezara a ejecutar el operador que se le halla indicado.

Debemos agregar un aspecto interesante, que es que los nodos no tienen porque enviarse a cualquier maquina o arquitectura de la red, sino que mediante las funcionalidades de PVM esto puede controlarse. La herramienta hace uso de este tipo de funcionalidades para ofrecer cosas como que en la definición de los nodos podemos utilizar una cláusula "on machine {nombre de maquina}" u "on architecture {nombre de la arquitectura".

Continuando con el tema de operadores, si el operador que se indico es un "**READ OPERATOR**" se abrirá el archivo correspondiente y como se explicara a continuación se enviaran los datos a los nodos consecutivos. Si el operador es "**WRITE OPERATOR**" lo que se valla recibiendo se ira colocando en el archivo que se indico en la definición del operador. El caso de "**IDENTITY OPERATOR**" es si se quiere aun mas sencillo, simplemente cuando algo llega se reenvía.
El caso un poco mas complejo es el de "**AGGREGATE OPERATOR**". En este caso todo lo que se recibe, es colocado en un vector donde si se constata que el elemento ya existía (mediante igualdad en los campos fijos) se le aplica la operación de agregación que se dio en la definición del operador (esto lo veremos en "Implementación del AGREGATE OPERATOR (P9)") a los campos que corresponda. Luego de haber terminado de realizar toda la agregación recién se comienza entonces a propagar los datos calculados.

[6] Ver en el apéndice un ejemplo completo.

El aspecto que nos resta es la comunicación entre procesos, para esto se implemento la idea de RIVER[7], esto explicado en pocas palabras es un ambiente de comunicación en el cual unos procesos colocan datos utilizando lo que se define como un "*sink"* y sacando datos de lo que se llama un "*source*".
Cada operador recibe internamente para trabajar todos los "*sources*" de donde obtener datos y los "*sinks*" donde volcarlos. Obviamente hay excepciones, el operador de lectura no recibe nada por lo que no se les pasan los "*sources*", así mismo el de grabación no recibe los "*sinks*".

Los "*sources*" y los "*sinks*" en la implementación en definitiva no son mas que puntos de acceso a las rutinas de PVM, enmascaradas con estos nombres. Cuando se coloca en un "*sink*" un registro simplemente se empaquetan con PVM todos los campos del registro y se le envían al proceso receptor, pero este sin embargo los ve como saliendo de un "*source*" que internamente realiza la operación inversa, es decir recibe y des-empaqueta.

La implementación también para ser un poco mas viable agrega el hecho de que arma mensajes mas grandes que un registro al momento de enviar, y en el caso de un nodo que envía el mismo dato a múltiples nodos utiliza multicast en lugar de envíos simples.

Implementación del AGREGATE OPERATOR (P9)

Este es el operador que levanta el valor de la aplicación, no porque su implementación sea novedosa o distinta sino porque abandona la línea de lo poco practico para entrar en el área de resolución de problemas reales como mostraremos en el ejemplo.

Para refrescar lo que hace esta primitiva mencionaremos que básicamente dada una relación de medida[8], esta genera otra, en la cual la información es resumida (o agrupada) por un conjunto dado de atributos.
La entrada de esta primitiva según se define en [AM2000] es:

- source schema : R $(A_1, ...,A_n) \in Rel_M$[9]
- Z, conjunto de atributos / card(Z) = k (medidas)
- $\{e_1, ..., e_k\}$, expresiones de agregación
- $Y / Y \subset \{A_1, ..., A_n\} \wedge Y \subset (Att_D(R) \cup Att_M(R))$, atributos que serán removidos
- instancia origen : r

Y en la implementación simplemente se cambia levemente el conjunto Y que se toma por los atributos que permanecen en vez de los que se remueven (solamente por un tema de facilidad al momento de implementar).

La sintaxis concreta con la que se da el comando es[10]:
'agregate operator' '(' [{Atributos} / record format '(' {Atributos de RF} ')'] ';' {Atributos a Agregar} ')'
Donde:

- 'Atributos' Es una lista de identificadores de atributos con la forma: {Atributo} [',' {Atributo}]*, donde:
  - 'Atributo' Es un atributo definido en el formato de registro de entrada o salida del nodo.
- 'Atributo de RF' Es idéntico a 'Atributos' salvo que cada atributo en realidad es el nombre de un formato de registro.
- 'Atributos a Agregar' Es una lista de identificadores de atributos con la forma: '(' {Atributo}',' {Agregación} ')' [',' '(' {Atributo}',' {Agregación} ')']*, donde:
  - 'Atributo' Es el definido anteriormente y
  - 'Agregación' es uno de los posibles métodos de agregación (p.e. 'sum'), use 'list agregators' para mas información.

[7] De ahí el nombre de la herramienta.
[8] El conjunto de las "measure relations" son las "crossing relations" que tienen al menos un atributo de medida. A su vez el conjunto de las "crossing relations" son las relaciones que representan relaciones o combinaciones entre los elementos de un grupo de dimensiones.
[9] Rel_M es el conjunto de relaciones de medida.
[10] Dentro de la aplicación esta disponible con 'help define nodes agregate operator', por mas información referirse al .

El Ejemplo

Para mostrar la aplicación en funcionamiento lo que se planteo fue seleccionar un problema en concreto, y plantearse utilizar la herramienta para resolverlo. La elección del problema no fue para nada trivial, dado que para mostrar la herramienta no había mucho problema, pero si en realidad se desea hacer interesante y que la versión paralela tenga posibilidades de ser mas eficiente que la no paralela no puede elegirse cualquiera.

El problema que se atacara es el de generación de redundancia para el tema de análisis OLAP. Para esto se planteo un problema típico de ejemplo que en nuestro caso pasa por el análisis de ventas en función de la cantidad vendida y el importe. Para analizar esta información se plantearon como dimensiones: el supervisor del vendedor que realizo la venta, el cliente que la compro, el producto que compro y en que fecha lo hizo. Estas dos ultimas dimensiones además con jerarquías conformadas por grupo de producto, tipo de producto y producto en el primer caso y año, mes y día en el segundo. El siguiente esquema estrella intentara mostrar el problema gráficamente:

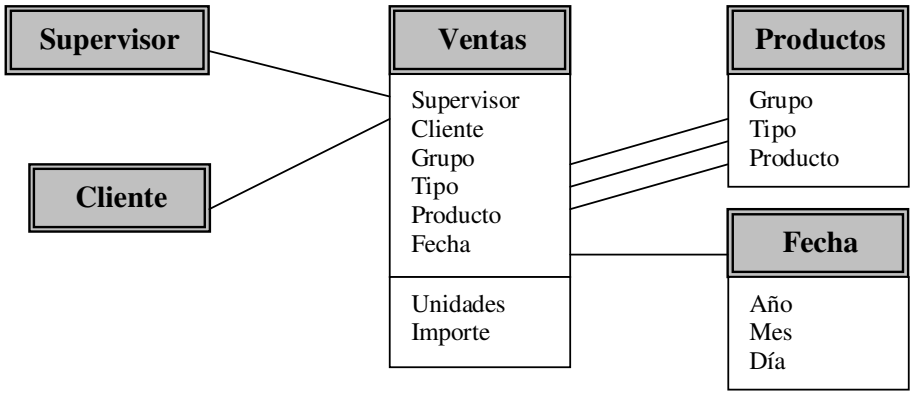

Figura 4: Diagrama estrella del ejemplo propuesto.

Repasando lo que queremos hacer (ahora ya con el diagrama), se trata de generar las agregaciones necesarias para tener pre-calculada la redundancia de los posibles puntos de análisis sobre el diagrama anterior. Estos son los distintos cruzamientos de las dimensiones antes presentadas.
Por ejemplo, una opción puede ser, mirar todos los supervisores, analizando todos los clientes y viendo todos los productos (sin jerarquizar por grupo y tipo) por año. Esto involucra agregar la información[11] por año, es decir eliminar el detalle por meses y días de los datos originales.

Sin tomar en cuenta las jerarquías de las dimensiones Producto y Fecha, estimando también que de las fechas conviene tomar como nivel inferior de detalle los años nos queda para analizar posibilidades en función de cuatro variables S (Supervisor), C (Cliente), P (Productos), A (Año)[12].

Ahora nuestro problema pasa a como plantear todos nuestros posibles cruzamientos de dimensiones (es decir donde calcular redundancia). Bueno una estrategia puede ser la siguiente: dadas cuatro variables en juego si las colocamos todas significara que estamos analizando las cuatro disgregadas, es decir sin resumir ninguna de las variables (si obviamente de los datos de entrada ya estamos resumiendo dado que el nivel de detalle era superior, pero para simplificar no colocaremos una agregación aquí). Esto podría ser un primer objetivo (SCPA) en nuestro problema donde todas estas variables se encuentran sin manipular.
El siguiente paso podría ser resumir una sola de ellas a la vez y mantener las otras intactas, lo que daría origen a: CPS, SPA, SCA y SCP. Así podríamos ahora hacerlo con dos por lo que tendríamos: CP, SA, PA, SC, CA, SP y finalmente tres: C, P, A, S.

Si miramos un poco en detalle esto ultimo vemos que nos lleva a trabajar con combinatorias quedándonos la formula: $C^4_4 + C^4_3 + C^4_2 + C^4_1 = 1 + 4 + 6 + 4 = 15$ que nos da la cantidad de cruzamientos posibles para este caso[13].

[11] De aquí la utilización de la primitiva de agregación (P9).
[12] Esto se realizo dado que era inmanejable afrontar el problema con mas de estas variables para el marco de este trabajo, y si se hubiera hecho así su aporte en relación no seria significativo.

Ahora el otro problema que enfrentamos es como organizar el grafo del proceso para realizar los cálculos. Lo primero que podemos ya intuir es que cada una de las agregaciones se realiza en un nodo utilizando el operador de agregación que presentamos en anteriormente, ahora como conectamos estos nodos?, los conectamos?.

Una alternativa es un grafo totalmente desconexo (obviando los nodos de lectura y escritura) que representaría que cada nodo lee los datos realiza la agregación y graba el resultado. Rápidamente podemos deducir que hay una forma aparentemente mas inteligente de reaprovechar el trabajo de los otros nodos, mirando el problema global de calcular todas las redundancias seria hacer una **partición funcional** del problema. Esto da entrada a nuestra primera propuesta de grafo:

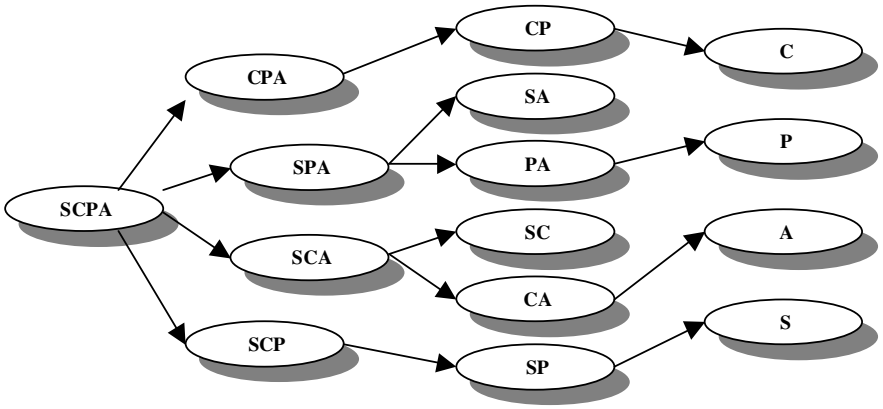

Figura 5: Grafo del proceso con partición funcional (pipeline)

Otra posibilidad que también rápidamente se nos puede venir a la mente es aplicar la técnica de **partición de dominio** esto como veremos en las líneas futuras podría facilitarse ampliamente, pero no se incorporo para la presente versión. La idea a seguir es simple, se pueden partir los datos del dominio de entrada y crear múltiples nodos para resolver el problema y luego juntar el resultado.
En nuestro ejemplo no tiene diferencia a los efectos del resultado final calcular sub-agregaciones y luego realizar una agregación final, pero a los efectos prácticos esto puede tener ventajas de performance porque los volúmenes de datos para cada nodo son mas manejables.

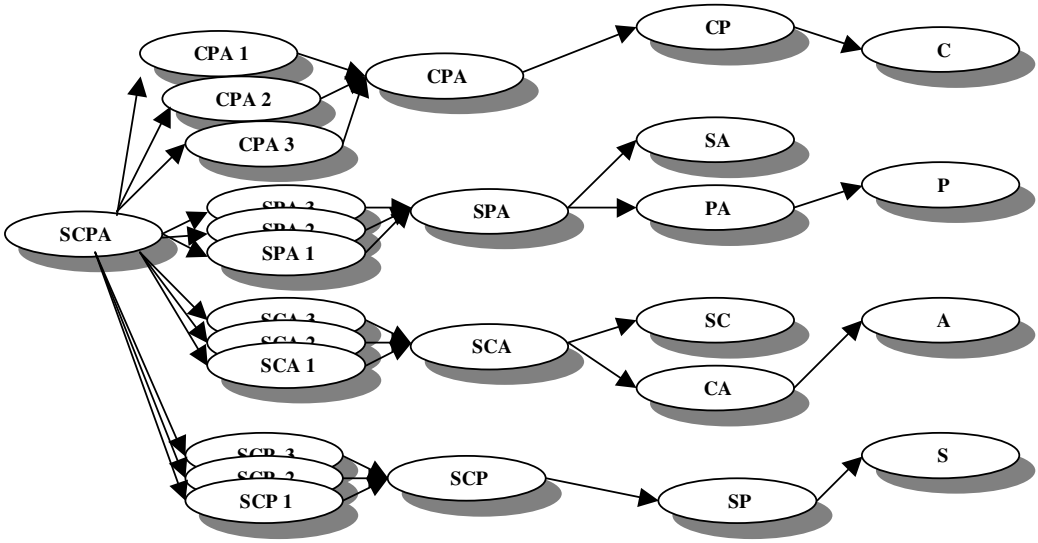

Figura 6: Grafo alternativo para el problema con partición de dominio.

[13] Esto de una manera similar salvo que eliminando situaciones inconsistentes por el hecho de tener jerarquías nos hubiera permitido poner una cota superior en la cantidad de casos del problema original en el orden de los 255 casos que hacia inmanejable el problema.

Para redondear fuera de cual sea el grafo que se elija para ejecutar la solución (dado que ambos producen el mismo resultado), agregándole el nodo de lectura y los de grabación correspondientes tendremos nuestra solución para resolver el ejemplo.

Líneas de Trabajo Futuro

Podemos presentar varias líneas de trabajo, muchas de ellas como veremos si bien se planteaban en [BBGS] o en [MG2000] escaparon al alcance de este trabajo. Entre ellas tenemos:
Nuevos operadores: Contamos con cuatro operadores (lectura, escritura, identidad y agregación) la mayoría de estos fueron considerados básicos para cerrar un primer acercamiento a la solución dentro del alcance del proyecto, pero claramente se podría trabajar por ejemplo en implementar todas las primitivas que se presentan en [AM2000] con lo cual se aumentaría considerablemente el poder de la herramienta.
Particiones de dominio automáticas: Uno de los punto en los que vimos se podía ampliar la herramienta de una forma muy practica trata el tema de facilitar en ciertos casos realizar particiones de dominio automáticas. Obviamente según el tipo de problema esta puede ser una solución viable o no, pero en caso que lo sea esto parece una buen característica que la herramienta podría incorporar.
Ejecución por maquina mono proceso: Cuando se plantea el grafo y en particular una serie de nodos se envían a ejecutar a una misma maquina, tenemos un overhead que podría ser fácilmente quitado realizando una ejecución de ese conjunto de nodos de forma centralizada, es decir dentro de un mismo proceso. Se trataría de hacer que ese sub-grafo actuara como un solo nodo hacia el resto del sistema.
Mejorar las estrategias de comunicación: Si bien se resolvió el problema de la comunicación de los nodos y el pasaje de información, e incluso se mejoro frente a una primera versión en la que se transmitía un registro por mensaje, es claro pensar que analizando mas en profundidad el tema se podría mejorar. Esto ultimo principalmente porque no se hizo mucho hincapié en esto al momento de resolverlo.

Conclusiones

Sobre los objetivos iniciales planteados de cobrar experiencia en la herramienta PVM a nuestro parecer esto se alcanzo, ya que se llegaron a manejar bastantes áreas (creación de procesos, comunicación, sincronización, etc.), e incluso se llego a pasar por esos detalles que uno no descubre hasta que le da problemas (por defecto se soportan 60 procesos por maquina aproximadamente) y encuentra solución en un recóndito lugar del manual.

Sobre el interés de hacerlo atractivo para el grupo de Concepción de Sistemas de Información se llego a implementar la primitiva P9 y usarla en forma practica para plantear solución a un problema real. Además potencialmente no parece irreal decir que el uso del paralelismo pueda mejorar procesos de carga similares a estos aunque no se tengan datos concretos, en este enfoque es probable que se halla cumplido el objetivo.

También se puede notar al leer las posibles líneas de trabajo que hay buen material por donde seguir si se quiere, e incluso algunos de los puntos parecen no estar lejos de poderse completar en un corto plazo.
Un aspecto que resulto interesante que vale la pena resaltar y que se constato luego de haber finalizado el desarrollo, fue la facilidad para cambiar de estrategia al momento de diseñar el proceso de transformación y así rápidamente poder probar otra nueva alternativa para resolver el mismo problema.

Por otro lado a nuestro entender el trabajo esta incompleto sin un análisis de los tiempos de ejecución, no parece razonable plantear una herramienta paralela para intentar mejorar los tiempos de ejecución; pensar un ejemplo que tuviera oportunidad de mejorar estos tiempos. Y finalmente quedarse sin poder ofrecer una respuesta en este tema, tanto así la respuesta fuera que la solución paralela es mejor, o en el pero caso, argumentar que la solución paralela no aporta valor.
Frente a esto ultimo la respuesta es que aun no se ha podido conseguir el equipo necesario para realizar las pruebas pero es la intención incluirlas en el documento.

Apéndice – Ejemplo completo de un proceso

Presentaremos aquí un ejemplo completo que se menciono en el documento y que fue utilizado como prueba de la aplicación. Ya anteriormente se presento el diagrama estrella de análisis que se realizo pero no el formato de todos los datos involucrados[14].

Nombre de Campo	Tamaño	Tipo
Supervisor	10	String
Cliente	10	String
Grupo	10	String
Tipo	10	String
Producto	10	String
Día	2	Integer
Mes	2	Integer
Año	4	Integer
Unidades	10	Integer
Importe	10	Float

Tabla 1: Tabla de definición de atributos del ejemplo.

Lo primero que presentaremos serán los formatos de registro necesarios para intercambiar la información de los nodos[15]:

```
define record format FormatoDeMedidas { Units,10,Integer; Amount,10,Double };

define record format S { Supervisor,10,String };
define record format C { Client,10,String };
define record format P { Group,10,String; Type,10,String; Product,10,String };
define record format A { Year,4,Integer };

define record format FormatoS extends S, FormatoDeMedidas;
define record format FormatoC extends C, FormatoDeMedidas;
define record format FormatoP extends P, FormatoDeMedidas;
define record format FormatoA extends A, FormatoDeMedidas;

define record format FormatoSC extends S, C, FormatoDeMedidas;
define record format FormatoSP extends S, P, FormatoDeMedidas;
define record format FormatoSA extends S, A, FormatoDeMedidas;
define record format FormatoCP extends C, P, FormatoDeMedidas;
define record format FormatoCA extends C, A, FormatoDeMedidas;
define record format FormatoPA extends P, A, FormatoDeMedidas;

define record format FormatoSCP extends S, C, P, FormatoDeMedidas;
define record format FormatoSCA extends S, C, A, FormatoDeMedidas;
define record format FormatoSPA extends S, P, A, FormatoDeMedidas;
define record format FormatoCPA extends C, P, A, FormatoDeMedidas;

define record format FormatoSCPA { Supervisor,10,String; Client,10,String; Group,10,String;
Type,10,String; Product,10,String; Day,2,Integer; Month,2,Integer; Year,4,Integer } extends
FormatoDeMedidas;
```

Como vemos es posible utilizar entre otras cosas la noción de herencia entre formatos de registro para facilitar su especificación. A continuación será necesario definir los nodos:

```
define nodes {
        NodoSCPA, read operator (W:\cprojects\data\Ventas-140249L.txt);

        NodoS, agregate operator (Supervisor; (Units, sum), (Amount, sum));
```

[14] Cabe resaltar que el formato presentado en la tabla será el que a continuación dará base a los formatos de registro definidos por eso se incluye especialmente.

[15] Código textual sacado de la distribución de la herramienta que se envía como ejemplo.

```
        NodoC, agregate operator (Client; (Units, sum), (Amount, sum));
        NodoP, agregate operator (Group, Type, Product; (Units, sum), (Amount, sum));
        NodoA, agregate operator (Year; (Units, sum), (Amount, sum));

        NodoSC, agregate operator (record format (S,C); (Units, sum), (Amount, sum));
        NodoSP, agregate operator (record format (S,P); (Units, sum), (Amount, sum));
        NodoSA, agregate operator (record format (S,A); (Units, sum), (Amount, sum));
        NodoCP, agregate operator (record format (C,P); (Units, sum), (Amount, sum));
        NodoCA, agregate operator (record format (C,A); (Units, sum), (Amount, sum));
        NodoPA, agregate operator (record format (P,A); (Units, sum), (Amount, sum));

        NodoSCP, agregate operator (record format (S,C,P); (Units, sum), (Amount, sum));
        NodoSCA, agregate operator (record format (S,C,A); (Units, sum), (Amount, sum));
        NodoSPA, agregate operator (record format (S,P,A); (Units, sum), (Amount, sum));
        NodoCPA, agregate operator (record format (C,P,A); (Units, sum), (Amount, sum));

        SaveNodoS, write operator (W:\cprojects\data\calculated\nodeS.txt);
        SaveNodoC, write operator (W:\cprojects\data\calculated\nodeC.txt);
        SaveNodoP, write operator (W:\cprojects\data\calculated\nodeP.txt);
        SaveNodoA, write operator (W:\cprojects\data\calculated\nodeA.txt);

        SaveNodoSC, write operator (W:\cprojects\data\calculated\nodeSC.txt);
        SaveNodoSP, write operator (W:\cprojects\data\calculated\nodeSP.txt);
        SaveNodoSA, write operator (W:\cprojects\data\calculated\nodeSA.txt);
        SaveNodoCP, write operator (W:\cprojects\data\calculated\nodeCP.txt);
        SaveNodoCA, write operator (W:\cprojects\data\calculated\nodeCA.txt);
        SaveNodoPA, write operator (W:\cprojects\data\calculated\nodePA.txt);

        SaveNodoSCP, write operator (W:\cprojects\data\calculated\nodeSCP.txt);
        SaveNodoSCA, write operator (W:\cprojects\data\calculated\nodeSCA.txt);
        SaveNodoSPA, write operator (W:\cprojects\data\calculated\nodeSPA.txt);
        SaveNodoCPA, write operator (W:\cprojects\data\calculated\nodeCPA.txt)
};
```

Aquí también pudimos ver otro aspecto que facilita la utilización que es utilizar la definición de formatos de registro para especificar los nodos que se desea mantengan en la agregación. Finalmente definiremos las aristas del grafo:

```
define edges {
        AristaSCPA-SCP, NodoSCPA, NodoSCP, FormatoSCPA;
        AristaSCPA-SCA, NodoSCPA, NodoSCA, FormatoSCPA;
        AristaSCPA-SPA, NodoSCPA, NodoSPA, FormatoSCPA;
        AristaSCPA-CPA, NodoSCPA, NodoCPA, FormatoSCPA;

        AristaSCA-SC, NodoSCA, NodoSC, FormatoSCA;
        AristaSCP-SP, NodoSCP, NodoSP, FormatoSCP;
        AristaSPA-SA, NodoSPA, NodoSA, FormatoSPA;
        AristaCPA-CP, NodoCPA, NodoCP, FormatoCPA;
        AristaSPA-PA, NodoSPA, NodoPA, FormatoSPA;
        AristaSCA-CA, NodoSCA, NodoCA, FormatoSCA;

        AristaCP-C, NodoCP, NodoC, FormatoCP;
        AristaSP-S, NodoSP, NodoS, FormatoSP;
        AristaPA-P, NodoPA, NodoP, FormatoPA;
        AristaCA-A, NodoCA, NodoA, FormatoCA;

        AristaSaveS, NodoS, SaveNodoS, FormatoS;
        AristaSaveC, NodoC, SaveNodoC, FormatoC;
        AristaSaveP, NodoP, SaveNodoP, FormatoP;
        AristaSaveA, NodoA, SaveNodoA, FormatoA;
```

```
        AristaSaveSC, NodoSC, SaveNodoSC, FormatoSC;
        AristaSaveSP, NodoSP, SaveNodoSP, FormatoSP;
        AristaSaveSA, NodoSA, SaveNodoSA, FormatoSA;
        AristaSaveCP, NodoCP, SaveNodoCP, FormatoCP;
        AristaSavePA, NodoPA, SaveNodoPA, FormatoPA;
        AristaSaveCA, NodoCA, SaveNodoCA, FormatoCA;

        AristaSaveSCP, NodoSCP, SaveNodoSCP, FormatoSCP;
        AristaSaveSCA, NodoSCA, SaveNodoSCA, FormatoSCA;
        AristaSaveSPA, NodoSPA, SaveNodoSPA, FormatoSPA;
        AristaSaveCPA, NodoCPA, SaveNodoCPA, FormatoCPA
};
```

Esto define los formatos de registro, nodos y aristas del grafo necesarios para correr un ejemplo y generar los archivos con las agregaciones calculadas. Cabe resaltar que los caminos ya sea del archivo origen y destino claramente deben adaptarse a la instalación en particular.

Una vez realizado esto y colocado correctamente el archivo de origen de datos (según lo especificado en el NodoSCPA) estamos en condiciones de utilizar el comando "**execute**" mencionado en el "Apéndice – Comandos de la Herramienta" si lo que deseamos es hacer correr la ejecución en forma distribuida o el comando "**execute alone**" si lo que queremos es que la ejecución se realice centralizada.

Apéndice – Comandos de la Herramienta

Los comandos disponibles en la aplicación son[16]:

- **define** : Define alguno de los elementos definibles del sistema (nodes, etc.).
  - Opciones: '**record format**', '**nodes**', '**edges**'.
- **list** : Lista los elementos ya definidos según el parámetro.
  - Opciones: '**record format**', '**nodes**', '**edges**', '**operators**', '**agregators**'.
- **help** : Presenta este help o el help particular de un comando u elemento.
- **quit** : Termina la aplicación.
- **file** : Lee el contenido de un archivo y lo ejecuta en el interprete.
- **execute** : Ejecuta en función de la información dada al sistema, en modo distribuido, es decir lanzando un proceso por operador en las distintas maquinas de la maquina virtual.
- **execute alone** : Ejecuta en función de la información dada al sistema, en modo autónomo, es decir un solo proceso que ejecuta toda la secuencia en esta maquina.

Referencias

- [BBGS] Tom Barclay, Robert Barnes, Jim Gray, Prakash Sundaresan, *Loading Databases Using Dataflow Parallelism.*
- [MG2000] Tobias Mayr, Jim Gray, *River Design*.
- [IL2001] Ignacio Larrañaga, *Análisis de Articulo: "Loading Databases Using Dataflow Parallelism"*
- [AM2000] Adriana Marotta, *Data Warehouse Design and Maintenance through Schema Transformations.*
- [PVM94] Al Geist, ,Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, Vaidy Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for networked Parallel Computing*

[16] Por mas información sobre la sintaxis de un comando en particular se recomienda usa el comando help mencionado aquí mismo.