PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 05-02

Relevamiento : Diseño Físico de Sistemas OLAP

Federico Piedrabuena Gustavo Vázquez

Abril de 2005

Relevamiento : diseño físico de sistemas OLAP
Piedrabuena, Federico; Vázquez, Gustavo

ISSN 0797-6410
Reporte Técnico RT 05-02
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, abril de 2005

Relevamiento: Diseño Físico de Sistemas OLAP

Federico Piedrabuena, Gustavo Vázquez
Instituto de Computación,
Universidad de la República,
Montevideo, Uruguay
{fpiedrab, gusvaz}@fing.edu.uy

Abstract. El diseño físico de los sistemas OLAP requiere técnicas completamente distintas al de los sistemas OLTP. Dado el gran volumen de información involucrado y las operaciones a las que está orientado un sistema OLAP, los métodos de diseño deben estar orientados a la eficiencia en la resolución de consultas. Para esto se requieren nuevas estructuras de datos y acceso, nuevas metodologías de diseño y procesos de mantenimiento eficientes. Este trabajo refleja el relevamiento de las actuales líneas de investigación en el diseño físico de sistemas OLAP e identifica algunos temas de investigación incipientes. Provee también un overview de las estructuras, arquitecturas, algoritmos, metodologías y tecnologías involucradas en el data warehousing.

Palabras claves: Data Warehourse, Diseño Físico, OLAP, Índices, Compresión, *VISS Problem*, Caché.

I. Introducción

El "*Online Analytical Processing*" (OLAP) requiere la ejecución de operaciones costosas, como por ejemplo *joins* y *aggregations*. Esta situación se hace mas compleja por el hecho de que las consultas OLAP deben realizarse sobre estructuras que tienen potencialmente millones de registros y porque los resultados tienen que ser entregados interactivamente al analista de negocios que opera el sistema. Dadas estas características, el énfasis en el ambiente *OLAP* está en el procesamiento eficiente de consultas.

En términos técnicos, el sistema *OLAP* es una proyección multidimensional redundante de una relación. Computa todos los g*roup by* (tomando operadores SQL como ejemplo) y realiza una agregación de sus resultados en un espacio N-dimensional para responder consultas *OLAP*. Estas agregaciones son entonces almacenadas en tablas de sumarización derivadas o arreglos multidimensionales. Dado que estas agregaciones generalmente son muy grandes y que las consultas son a menudo complejas se hace necesario acelerar los tiempos de respuesta de las consultas usando mejores modelos, estructuras de datos, índices y métodos de compresión.

Estas son algunas de las razones por las que el diseño físico de sistemas OLAP es diferente del diseño de bases de datos orientadas a sistemas OLTP. A esto, se le suma el hecho de que los sistemas OLTP requieren bajos tiempos de respuesta en operaciones del tipo *updates*, mientras que los sistemas OLAP buscan bajos tiempos de respuestas en operaciones de consultas. Asimismo, el cambio de perspectiva a un enfoque multidimensional [Kimball96] influye en la organización de datos y esta en las estructuras de acceso.

El principal factor a considerar en la etapa de diseño de un sistema OLAP es el gran volumen de información involucrado. Por esta razón las líneas de investigación en esta área apuntan al incremento en la performance tanto de la ejecución de consultas como de la carga inicial o incremental teniendo en cuenta la optimización en la utilización del espacio.

La investigación de este trabajo se ha centrado en ocho grandes áreas de investigación:

- Modelo de Datos: En esta área se estudian las estructuras de datos utilizadas en almacenamiento físico, así como los mecanismos de aplicación de estas según los requerimientos generales del sistema OLAP.
- Compresión: Dado el gran volumen de información manejado y los altos grados de dispersión involucrados, debido a que no todos los posibles cruzamientos de dimensiones determinan valores para las medidas, muchas de las estructuras estudiadas no optimizan el espacio de almacenamiento. Por esta razón se han desarrollado varios métodos de compresión que permiten consultar los datos sin descomprimirlos.
- Índices: El estudio de índices aplicables a los sistemas OLAP es un tema importante y extensamente investigado. Muchos de los índices ya existentes para sistemas OLTP son revisados y modificados para sistemas OLAP.
- Selección de vistas e índices: Una de las principales actividades en la etapa de diseño es la selección del conjunto adecuado de vistas e índices a materializar para obtener el mejor desempeño, respetando las restricciones de espacios y tiempos de mantenimiento. Este problema es conocido como el problema de selección de vistas e índices, *View and Index Selection Problem* (*VIS Problem*).
- Caché: Cuando no se conoce la frecuencia de ocurrencia de las consultas no es posible seleccionar un conjunto de vistas a materializar. Por lo tanto, se requieren técnicas de materialización dinámicas basadas en las condiciones de uso del OLAP. En estos casos, el caché es la mejor respuesta a esta problemática.
- Optimización de Consultas: Una vez aplicados los métodos de diseño anteriores, es necesario realizar cambios a las consultas realizadas a los sistemas OLAP. De esta manera, se podrán aprovechar los beneficios brindados por los métodos vistos, principalmente la materialización de vistas y el caché de respuestas.
- Fragmentación y Distribución: Tal como se estudia en el diseño de bases de datos para sistemas OLTP, muchas veces es necesario distribuir los datos entre distintos sitios para mejorar la performance y la disponibilidad.
- Carga y Mantenimiento: Dado el gran volumen de información involucrado, la complejidad de las estructuras de datos y el extensivo uso de índices, es necesario contar con mecanismos de carga y mantenimiento eficientes. La carga y el mantenimiento complementan muchas de las metodologías existentes de diseño físico con la finalidad de cumplir, de manera global, la optimización propuesta.

Estos temas son tratados en el trabajo. Para cada caso, se relevó la información relacionada y las investigaciones realizadas en el área. En la sección II se especifican las estructuras de datos; las técnicas de compresión en la sección III y en la sección IV se describen las principales estructuras de índices. En la sección V se especifican los problemas a resolver en la etapa de diseño. En la sección VI se describen las técnicas de caché y en la VII se describen técnicas de optimización de consultas. En la sección VIII se enumeran las posibles arquitecturas enfocadas principalmente a la fragmentación y distribución de datos y finalmente, en la sección IX, se presenta el tema de carga y mantenimiento de sistemas OLAP. La sección X muestra algunas consideraciones que deben tomarse en cuenta a la hora de seleccionar cuál es el método de diseño físico más apropiado para el escenario sobre el que se aplicará. Finalmente, la sección XI mostrará algunas conclusiones extraidas del relevamiento realizado.

II. Modelo de Datos

Los Data Warehouses deben ser diseñados estructurando los datos de una manera que puedan ser manejados por el sistema *OLAP*. Generalmente, se encuentran dos técnicas para modelar los Data Warehouses: el modelo multidimensional y el modelo relacional. Estas dos técnicas proporcionan una vista multidimensional de los datos que soporta y facilita el acceso a los mismos por parte de las aplicaciones *OLAP*.

El modelo relacional es llamado *ROLAP* [Harinarayan96, Agrawal95] o *RDW* (Relational Data Warehouse). El otro modelo es llamado *MOLAP* [Harinarayan96, Agrawal95] o *MDW*

(Multidimensional Data Warehouse). Los *RDW*s se construyen sobre un manejador de bases de datos relacionales (*RDBMS*), mientras que los *MDW*s se basan en manejadores multidimensionales de base de datos (*MDDBS*). Las estructuras de datos en las cuales los *RDW*s y *MDW*s almacenan sus datos son radicalmente diferentes como se podrá ver en las siguientes secciones.

La estrategia relacional es muy escalable y puede manejar Data Warehouses muy grandes; sin embargo este crecimiento afecta la performance de las consultas. El enfoque multidimensional, por otra parte, tiene mucho mejor desempeño en el procesamiento de consultas, pero no es muy escalable. Ésas son las razones de la existencia de estas dos estrategias y lo que fundamenta su investigación.

En los últimos tiempos, con el advenimiento de la Web, han surgido otras técnicas que utilizan tipos de datos semiestructurados tanto como fuente de datos, repositorio información multidimensional y mecanismo de intercambio. Es interesante ver la capacidad de estos tipos de datos para la implementación de Data Warehouses y los trabajos que existen al respecto.

II.1 ROLAP

Los RDWs utilizan tablas relacionales como estructura de datos, es decir, una "celda" en un espacio lógico multidimensional se representa como un registro con algunos atributos que identifican la ubicación de la celda en el espacio multidimensional y otros que contienen los valores de las medidas de la celda.

Esta estrategia permite a los usuarios consultar directamente los datos en bruto, es decir, las operaciones en el *OLAP* se traducen en consultas relacionales. El problema de la performance de las consultas se ataca usando índices y otras estrategias convencionales de optimización de consultas relacionales.

A menudo, un *Data-Cube* se utiliza para modelar un Data Warehouse y una base de datos relacional se utiliza para su implementación. Un *Data-Cube* [Gray96, Ester01] consiste en varios atributos independientes, agrupados en dimensiones, y algunos atributos dependientes llamados medidas. Un *Data-Cube* puede verse como un arreglo d-dimensional cuyas celdas contienen las medidas para los respectivos sub-cubos.

En estrategias tradicionales de procesamiento de consultas que se basan en *Data-Cubes*, los sub-cubos de estos pueden ser precalculados y almacenados. Dada una consulta, se genera una búsqueda en el *Data-Cube* para obtener la información relevante. Los resultados de la búsqueda pueden requerir ser procesados, dependiendo del tipo de consulta, para finalmente retornar una respuesta exacta.

Dado el gran volumen de almacenamiento requerido por los modelos relacionales, debido al alto nivel de redundancia implicada en pos de buenos tiempos de respuesta, es que han surgido varias técnicas que permiten optimizar el espacio consumido sin perjudicar los tiempos de repuesta.

Es así que surgen los *Cuasi-Cubes* [Barbara97, Vitter99] que proveen una descripción incompleta del *Data-Cube* y un método de estimación de las entradas que falten con un cierto nivel de exactitud. La descripción debe tomar una fracción del espacio total del cubo y el procedimiento de estimación debe ser más rápido que el procesamiento de los datos que se encuentran en las tablas subyacentes.

La idea mostrada en [Barbara97] de *Cuasi-Cubes* es dividir el cubo en regiones y utilizar un modelo de estimación para describir cada región. El modelo proporciona una descripción de cada región, ocupando este menos espacio que el grupo de celdas modeladas. El modelo es utilizado para estimar las celdas de la región, lo que introduce errores (con respecto a los valores

originales). Para estimar las entradas que faltan, con un nivel razonable de exactitud, se utiliza regresión lineal que permite mantener acotado el error máximo. La regresión y la estimación del error son operaciones altamente paralelizables, lo que permite alcanzar mejores tiempos de respuesta. Para mantener el número de errores acotado, los valores de las celdas que producen los mayores errores son almacenados y no modelados.

En [Vitter99] se presenta un método que proporciona respuestas aproximadas basadas en *multi-resolution wavelet decomposition* que obtiene una representación aproximada de los datos subyacentes, especialmente cuando la información en bruto es muy dispersa.

[Roussopoulos97] describe una abstracción del almacenamiento del *Data-Cube* llamada *Cubetree*. Este se realiza con una colección bien organizada de *R-tree* empaquetados, que alcanzan altos grados de concentración de datos y bajos niveles de consumo de espacio.

Productos

Hay muchos productos que se basan en implementaciones ROLAP. Algunos de ellos son BusinessObjects [BOS], Microstrategy's DSS Agent [MIC], Redbrick [RED], Oracle Warehouse [ORA] y DB2 Data Warehouse [DB2]. La performance es mejorada materializando vistas del cubo y creando índices sobre ellas, lo que puede consumir mucho espacio.

II.2 MOLAP

Los *MDW*s almacenan sus datos mediante arreglos multidimensionales, guardando solamente los valores de las medidas ya que los valores de las dimensiones se tratan como índices de los arreglos multidimensionales. La posición de los valores de la medida dentro de los arreglos multidimensionales se puede calcular a partir de los valores de la dimensión.

Esta estrategia evita el uso de *SQL* y bases de datos relacionales, utilizando sistemas propietarios de base de datos multidimensionales (*MDDB*) y *API*s para OLAP. Mientras que la información en bruto está en Data Warehouses relacionales, el cubo se materializa en una *MDDB*, es decir, mantiene los datos como una matriz k-dimensional basada en una estructura especializada no relacional de almacenamiento. El diseñador de la base de datos especifica todas las agregaciones que considera útiles. Mientras se construyen las estructuras de almacenamiento, las agregaciones asociadas a todos los *roll-ups* posibles se precalculan y se almacenan. Así, los *roll-ups* y *drill-downs* se responden en tiempos pequeños, permitiendo por lo tanto consultas interactivas. Los usuarios consultan el cubo y el *MDDB* recupera eficientemente el valor de una celda a partir de su dirección. Generalmente se utiliza un esquema de *hashing* sobre la dirección de cada celda para asignar espacio solamente a las celdas presentes con información en bruto y no a cada posible celda del cubo.

Por ejemplo, en la estructura *Cube* los datos se extraen de las base de datos, se convierten en arreglos multidimensionales y se agrupan de modo que las consultas comunes del cubo requieran un I/O mínimo.

Productos

Arbor's Essbase [ARB], IRI Express [IRI], O3 [ISF], Microsoft Analysis Services [MSS], Cognos [CGN] y muchos otros MDDBs son implementados de esta manera. Esta línea materializa todas las celdas del cubo presentes en la información en bruto, lo que también requiere mucho espacio.

II.3 Semi-Estructurado

Fuente de Datos

Muchos trabajos se han realizado en el mundo académico para idear metodologías de diseño de sistemas de Data Warehousing capaces de integrar la información de diversas fuentes, en particular de bases de datos heterogéneas. Debido a que más organizaciones ven la web como parte integral de su estrategia de comunicación y negocio, la necesidad de integrar datos desde

documentos *XML* en los Data Warehouses cobra cada vez más importancia. Algunas herramientas comerciales soportan la extracción de datos de fuentes *XML* para alimentar el Data Warehouse, pero el esquema de este último, así como el mapeo lógico entre el esquema fuente y destino, deben ser definido por el diseñador.

Un enfoque alternativo a diseñar desde fuentes *XML* consiste en traducir estas primero a un esquema relacional equivalente, para luego comenzar de este último a diseñar el Data Warehouse. Existen varios trabajos acerca de como traducir documentos *XML* en base de datos relacionales, pero pocos tratan el problema de determinar la cardinalidad de las relaciones, lo que juega un papel importante en el diseño multidimensional.

En particular surgen dos temas cuando se trata este problema. Uno, la existencia de diversas técnicas para representar relaciones en *XML* (en particular, DTDs y esquemas), cada una con distinto poder de expresividad; el otro, dado que los *XML* modelan datos semiestructurados no toda la información necesaria para el diseño puede ser derivada con seguridad.

[Golfarelli01b] propone un enfoque semiautomático para la construcción del esquema conceptual de un Datamart partiendo de las fuentes *XML* y muestra cómo el diseño multidimensional para los Data Warehouses puede ser realizado directamente sobre una fuente *XML*. Su contribución es doble: por un lado, propone una revisión y comparación orientadas a Data Warehouses, de los enfoques para estructurar documentos *XML*; por el otro, propone un algoritmo en el cual el problema de deducir correctamente que información es necesaria, se soluciona consultando los documentos *XML* de las fuentes y, en caso de ser necesario, pidiendo ayuda al diseñador.

Repositorio Multidimensional

Otro enfoque en el estudio de los tipos de datos semiestructurados es el uso de los mismos para la implementación del Data Warehouse.

En [Hümmer03] se introduce *XCube*, una familia de plantillas basadas en documentos *XML* para almacenar, intercambiar y consultar datos de Data Warehouses. *XCube* se organiza modularmente, de manera que el esquema multidimensional, las descripciones de las dimensiones y los datos en sí mismos se pueden transmitir en pasos separados. Las ventajas están dadas por el uso de estándares, las desventajas por el hecho de que los documentos XML tienden a ser muy grandes. Los *XCube* consisten en un sistema de tres esquemas XML responsables de expresar el esquema multidimensional, las dimensiones y los valores de la tabla de medidas: *XCubeSchema*, *XCubeDimension* y *XCubeFact*. Estos tres esquemas permiten describir totalmente un *Data-Cube*, con el *XCubeSchema* conteniendo el esquema multidimensional, el *XCubeDimension* la estructura jerárquica de las dimensiones implicadas y el *XCubeFact* los datos de la tabla de medidas, es decir las celdas del cubo. Una razón para esta descomposición es la posibilidad de reutilizar algunos de estos documentos: por ejemplo un documento de *XCubeDimension* puede ser compartido por varios cubos o incluso aplicaciones. Otra razón es la posibilidad de minimizar la diversidad de terminología multidimensional.

Otra propuesta de estándar para metadata de Data Warehouses es *MetaCube-X* [Nguyen01] que también se basa en *XML*. Semejantemente a *CWM* se concentra principalmente en la metadata y no en los datos de la tabla de hecho, por lo que no hay separación entre el esquema y los datos de la dimensión.

III. Compresión

Las tablas de medidas o vistas materializadas se utilizan para mejorar los tiempos de proceso de consultas, especialmente las que implican grandes agregaciones de datos y costoso *joins* de varias tablas. Esto implica espacios de almacenamiento muy grandes, y generalmente altos grados de dispersión, lo que ha hecho de la compresión de datos una herramienta muy importante y efectiva.

Existen varias razones para la necesidad de comprimir datos en un Data Warehouse [Srivastava02]. La primera razón es que los conjuntos multidimensionales de datos creados por el producto cartesiano de las dimensiones pueden ser muy dispersos. La segunda razón es la necesidad de comprimir las descripciones del espacio multidimensional. A estas razones se le suma las características intrínsecas a los valores de los datos. A menudo estos se sesgan a conjuntos con muchos valores cercanos y muchos valores dispersos.

III.1 Propiedades y Requerimientos

Las técnicas normales de compresión pueden alcanzar grandes niveles de compresión, pero no son usables en este contexto, según lo mostrado en [Furtado00], con excepción del archivado de datos a nivel de filesystem. Esto es porque los datos deben ser comprimidos de a bloques, perdiendo toda capacidad de ser consultable, degradando la performance e imposibilitando la búsqueda por patrones sin incurrir primero en costosas descompresiones de bloques.

Hay varios requerimientos que debe cumplir una buena estrategia de compresión. Algunos de ellos son:

- devolver los resultados exactos o lo más aproximados posible para cualquier consulta
- comprimir los datos con eficacia
- integrarse al contexto del Data Warehouse
- ser aplicable a cualquier conjunto de datos de una tabla de medidas
- brindar tiempos de descompresión iguales o mejores que los tiempos de lectura de los discos para las exploraciones secuenciales

Pero los requerimientos principales son la capacidad de mantenimiento de los elementos comprimidos y la posibilidad de ser consultados sin ser descomprimidos, todo esto con buenos niveles de performance.

Es deseable entonces, desarrollar técnicas de compresión de datos que permitan consultar los datos en su forma comprimida y las operaciones se puedan realizar directamente en los datos comprimidos. Tales técnicas proporcionan generalmente dos mapeos. Uno llamado *forward mapping* que calcula la ubicación en el conjunto comprimido de datos dada una posición en el conjunto original. El otro llamado *backward mapping* que calcula la posición en el conjunto original dado una ubicación en el conjunto comprimido. Un método de compresión es *mapping-complete* si proporciona *forward mapping* y *backward mapping*. Los métodos de compresión *mapping-complete* son generalmente usados para bases de datos multidimensionales. Muchas técnicas de compresión son *mapping-complete*, como ser la *header compression*, *BAP compression* y *chunk-offset compression*.

Un procedimiento de compresión para *MDW*s se muestra en [Srivastava02] y consiste en primero almacenar cada conjunto en un arreglo multidimensional según los valores de la dimensión. Luego, este es transformado en un arreglo linearizado por medio de una función adecuada. Finalmente, el arreglo linearizado es comprimido por un método de compresión *mapping-complete*.

Un procedimiento de compresión para *RDW*s se muestra en [Goldstein98], en él un registro comprimido se puede identificar por un *page-id* y un *slot-id* de la misma manera que los registros sin comprimir son identificadas en un *DBMS*s convencional. La manera en que se utiliza el identificador de registro no cambia con la compresión.

III.2 Técnicas

En [Furtado00] se presenta *FCompress,* una técnica de compresión de información que mantiene la propiedad de consulta sin necesidad de descomprimir los datos. Esta técnica es aplicable a los datos de las tablas de medidas, sumarización y cubos. Está diseñado para integrarse al ambiente

del Data Warehouse y se basa en el reemplazo de los datos originales por valores aproximados de los atributos, almacenando *bitcodes* compactos.

[Goldstein98] introduce un algoritmo de compresión/descompresión a nivel de página de datos en estructuras relacionales. Explora también el uso de ordenamiento tipo *B-Tree* sobre datos comprimidos así como sobre los datos originales.

Los niveles de descompresión mostrados en [Goldstein98] son más rápidos que los tiempos de lectura típicos de los discos para exploraciones secuenciales. Este algoritmo es simple, y puede agregarse fácilmente a la capa de administración de archivos de un *RDBMS*, puesto que soporta la técnica usual de identificar un registro por el par (*page-id*, *slot-id*). Puede descomprimir registros individuales sin necesidad de descomprimir toda la página. También mejora las estructuras de índice *B-trees* y *R-trees* notoriamente reduciendo el número de páginas de hoja y comprimiendo entradas del índice. Este algoritmo tiene la desventaja de que comprime solamente campos numéricos, sin embargo, las tablas de medidas en Data Warehouses contienen muchos campos numéricos y de baja cardinalidad, y muy poco de texto.

En [Srivastava02] se presenta un conjunto de algoritmos de agregación en Data Warehouses comprimidos para OLAP multidimensionales. Estos algoritmos funcionan directamente sobre los conjuntos comprimidos mediante algún método *mapping-complete*, sin la necesidad de descomprimirlos. También se brinda un procedimiento de decisión, para seleccionar el algoritmo de agregación más eficiente, basado en la consulta y memoria disponible, así como también el conjunto de parámetros dados para la consulta.

IV. Índices

Uno de los principales temas de investigación en el diseño físico se refiere a la selección óptima de índices [Harinarayan97], que se basa en el esquema lógico y en la carga de trabajo, y requiere que se tomen en cuenta estructuras específicas de acceso. Los índices, por lo tanto, juegan un papel crucial en la performance del OLAP.

Los pasos para agregar índices van de técnicas simples, como índices secundarios, a estructuras complejas, como las que se mostrarán mas adelante. En general se piensa que el mejor subconjunto de índices es el que reduce al mínimo el costo de acceso. Si bien en la mayoría de los casos cualquier estructura de índice resuelve los requerimientos de performance, el problema es el costo asociado. Cuando el diseñador tiene la posibilidad de poner unos o más índices en una relación para mejorar ciertas consultas, la ventaja de materializar una vista puede ser afectada por el espacio que, invariablemente, dicho índice va a utilizar. Además del costo en espacio se agrega el tiempo de creación, administración de datos y overhead de algunas operaciones.

IV.1 Propiedades y Requerimientos

Dado que los sistemas OLAP manejan enormes volúmenes de información se hace necesario contar con estructuras de índices eficientes, que consuman poco espacio y que su creación y mantenimiento consuman pocos recursos.

Puesto que los Data Warehouses son generalmente actualizados *off-line*, los índices se pueden reorganizar periódicamente de forma de agrupar sus entradas óptimamente. Por lo tanto, no es necesario realizar el mantenimiento de los índices cuando ocurren actualizaciones de datos, lo que permite adoptar por estructuras más complejas sin que esto repercuta negativamente en los tiempos de respuesta de las consultas realizadas.

[O'Neil97] indica que puede ser deseable, en un ambiente de Data Warehouses, tener más de un tipo de índice disponible en una columna para poder elegir el mejor para cada consulta.

[Zhuge95] indica a su vez, que el proceso de creación de índices no debe consumir demasiado tiempo, de otra forma la operación se podría ejecutar más eficientemente sin el mismo. Es decir, el índice puede no ser particularmente útil si la ejecución de la operación sin el índice es varias veces más rápida que el tiempo total de ejecutarla, incluyendo el tiempo de construir el índice. Por supuesto, los índices se utilizan a menudo, aunque consuma mucho tiempo en su creación, si los datos indexados se consultan repetidas veces. En este caso, el tiempo de construir el índice es amortizado por el número de consultas hechas sobre los datos indexados.

Al indexar columnas de una tabla de medidas, almacenar el índice y la columna correspondiente en la tabla de medidas resulta en una duplicación de datos. En tales situaciones, puede ser recomendable solo almacenar el índice si los registros originales de la tabla se pueden reconstruir fácilmente a partir del índice, según lo presentado en [Datta99].

IV.2 Estructuras

En general un índice es cualquier estructura de datos que toma como entrada cierta propiedad de los registros, típicamente los valores de uno o más campos[1], y permite encontrar de forma rápida los registros que la cumple.

Los índices se clasifican en unidimensionales o multidimensionales en base al número de campos que interviene en él.

- Los índices unidimensionales son aquellas estructuras de acceso que aceptan un único *search-key* y en base a los valores de este retornan los registros correspondientes.
- Los índices multidimensionales son aquellas estructuras de acceso que aceptan más de un *search-key* y en base a los valores de estos retornan los registros correspondientes.

Si para resolver consultas multidimensionales se emplearan estructuras de índices unidimensionales para cada dimensión, el espacio en disco necesario y el número de entrada-salida requerido para obtener los punteros a los registros candidatos de cada dimensión sería tan grande que no producirían ventajas en la performance del sistema *OLAP*. Por este motivo es que en general se utilizan estructuras de índices multidimensionales. Estas se clasifican según su estructura subyacente en dos grupos, las basadas en árboles y las basadas en funciones de hash.

Basadas en Árboles

Un esquema simple basado en árboles puede verse como un índice de índices, o de manera más genérica, un árbol en el que los nodos de cada nivel son índices para una dimensión, como se muestra en la figura 1.

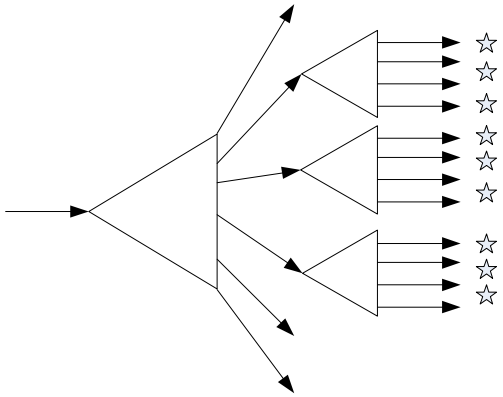

Figura 1: Estructura basada en árboles

[1] Los campos en los que se basa un índice son llamados *search-key* (clave de búsqueda).

El índice *B-tree* (creado por Bayer y McCreight en 1972) y sus variantes organizan los datos de una manera jerárquica y es, por lejos, la estructura de índice mas utilizada en base de datos relacionales.

Los índices *B-tree* [Gaede98] son árboles balanceados[2] en el que cada nodo interno contiene punteros a otros nodos correspondientes a intervalos, siendo generalmente las hojas las que contienen los punteros a los datos. Dependiendo del tipo de *B-tree* los nodos interiores podrían tener punteros a datos. Los *B-trees* tienen un límite superior e inferior para el número de descendientes de un nodo. El límite inferior previene la degeneración de los árboles y conduce a una utilización eficiente del almacenamiento. El límite superior sigue el hecho de que cada nodo del árbol corresponda exactamente a una página de disco.

Este tipo de índice es unidimensional, por lo que en el ambiente OLAP no es muy usado. No así sus variantes, como por ejemplo los *B+-Tree*, los *UB-Tree*, los *R-Tree* (junto con sus variantes), los *X-Tree* y los *DC-Tree*, que son multidimensionales.

El *B+-Tree* [Datta99, O'Neil97] es una variante del *B-Tree* y es muy utilizado en los sistemas relacionales para recuperar registros de una tabla a través de valores especificados que implican unas o más columnas. Cada entrada en el índice a nivel de hoja apunta a una lista de *RID*s[3] de registros que contienen los datos. Para cualquier *key-value* dado, esto es para cada valor distinto contenido en la columna de la tabla, la lista generada se almacena en una cadena de bloques de datos.

Una mejora significativa al *B+-Tree* está dada por los *UB-Tree* (Universal B-tree) [Fenkx00, Bayer97]. Se le llama así a cualquier variante de *B-tree* en el cual los *key-values* son direcciones de regiones ordenadas por cierta función. Las páginas hojas mantienen objetos o sus identificadores en regiones. Provee una performance logarítmica para operaciones básicas de inserción, borrado y consulta, y una utilización de página del 50% del original.

Una generalización del *B-Tree* está dada por *R-Tree* (creado por Guttman en 1984). [Ester01, Gaede98] generaliza el *B-tree* de 1-dimensión a espacios de datos d-dimensionales, es decir un *R-tree* maneja hiperrectángulos d-dimensionales en lugar de *key-values* numéricos 1-dimensiónales. Un *R-tree* puede organizar objetos extendidos, tales como polígonos, usando *MBR* (*Minimum Bounding Rectangles*) como aproximaciones, así como punteros a objetos como caso especial de rectángulos. Para resolver una consulta, primero se determina el conjunto de MBRs que responde la consulta empezando por la raíz y luego se buscan las referencias a sus nodos hijos hasta llegar a las páginas de datos. El *R+-Tree* (Sellis et al. 1987) [Gaede98] Es una variante del *R-Tree* en la que no se permite solapamiento entre las regiones.

Las inserciones aleatorias no solo son muy lentas, debido a la continua reorganización del espacio, sino que también destruyen los clusters de datos en todos los esquemas multidimensionales de indexación de direcciones. El *packed R-Tree* (creado por Roussopoulos en 1987) [Roussopoulos97] evita estos problemas, clasificando primero los objetos en cierto orden y cargando el *R-tree* desde el archivo, ordenado y empaquetando los nodos hasta su máxima capacidad. Este método de empaquetamiento ordenado alcanza excelentes niveles de clusterización y reduce significativamente la sobrecarga y espacios muertos.

X-Tree (creado por Berchtold en 1996) tiene ciertas ventajas respecto al *R-Tree* en los casos en que los valores son referenciados por una gran cantidad de dimensiones. El índice *X-tree* [Berchtold96, Ester01] se ha diseñado para trabajar eficientemente con valores grandes de dimensiones. Si la topología estándar divide los resultados con un alto grado de solapamiento, el *X-tree* intenta encontrar una división de mínimo solapamiento basada en la historia de divisiones. Si el número de elementos en una de las particiones resultantes está por debajo de un

[2] Significa que todos los caminos desde la raíz a las hojas tienen el mismo largo.
[3] RID: Identificador de Registro

umbral dado, la división será demasiado desequilibrada y, por lo tanto, el algoritmo de partición terminará sin producir división. En este caso, el nodo actual se amplía para convertirse en un supernodo múltiplo del tamaño de bloque estándar.

DC-Tree [Ester02] es un índice jerárquico estructurado similar al *X-tree* que explota el concepto de jerarquías definidas típicamente para las dimensiones de un cubo. El índice *DC-Tree* utiliza secuencias descriptivas mínimas y el orden parcial de los valores de los atributos inducidos por las jerarquías, en lugar de *MBR* y ordenamiento artificial total (*artificial total ordering*).

El índice *k-d-tree* [Gaede98] es unas de las más prominentes estructuras multidimensionales. Es un árbol binario de búsqueda que representa el universo d-dimensional de información por medio de subdivisiones recursivas en espacios de (d-1)-dimensiones. Cada nodo del árbol contiene un atributo y un valor, lo que permite dividir los hijos en aquellos que poseen valores mayores de los que poseen valores menores. En distintos niveles del árbol los nodos pueden hacer referencia a distintos atributos. La desventaja de estos índices es que la estructura es sensible al orden en que son insertados los punteros. Algunas variantes de este índice (como el *adaptative k-d-tree*, o el *bintree*), que pueden verse en [Gaede98], solucionan este problema.

El índice *Quad-Tree* [Ester01, Gaede98] es un índice donde las entradas de los nodos consisten en la descripción de un cubo secundario y el máximo valor de la medida materializado para este.

Basadas en Funciones de Hash

Las funciones de *hash* pueden recibir una lista de valores de atributos como argumento y retornar como resultado el puntero a la información solicitada. Esto hace atractivas las funciones de *hash* para ser utilizadas en estructuras de acceso. Generalmente se emplea una función de *hash* compuesta de *K* funciones de *hash*, cada una aplicable a una de las *K* dimensiones, como se muestra en la figura 2.

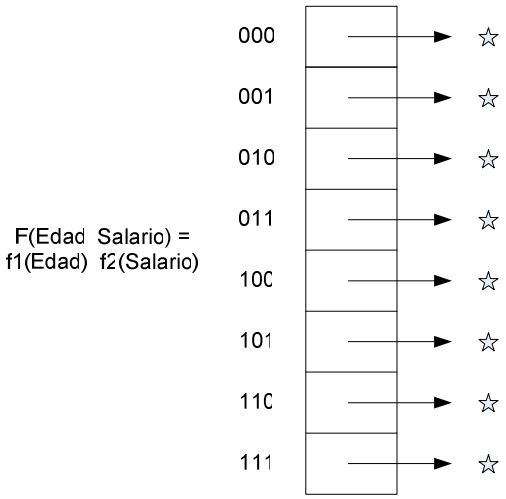

Figura 2: Estructura basada en funciones de hash

El *Linear Hashing Index* (creado por Larson en 1980) divide el universo [A,B) de posibles valores de *hash* en intervalos binarios de tamaños (B- A) / 2^k o (B- A) / 2^{k+1} para k > 0. Cada intervalo corresponde a un *bucket*. Un puntero t ∈ [A,B) separa los intervalos chicos de los grandes: los intervalos de tamaño (B- A) / 2^k van a la izquierda de t y los de tamaño (B- A) / 2^{k+1} a la derecha.

Otra opción es el *Budy Tree* (creado por Seeger y Kriegel en 1990), un esquema dinámico de *hash* con un directorio estructurado en forma de árbol. El árbol es construido por sucesivas inserciones, cortando el universo recursivamente en dos partes del mismo tamaño con hiperplanos iso-orientados.

Un caso particular de indice basado en hash es el *bitmap index* [Ester01, O'Neil95, O'Neil97, Datta99]. En ella, las páginas hojas de una estructura de índice no contienen listas de *RID*s, sino vectores de bits con un bit para cada registro. El bit tiene valor 1 en el *bitmap* si el registro contiene el valor en la lista representada; si no, el bit tiene valor 0. Esta técnica es particularmente atractiva cuando el conjunto de *key-values* posibles en el índice es pequeño y con una gran cantidad de registros, ejemplo un índice sobre un atributo género, donde GÉNERO = 'M' o GÉNERO = 'F'. Cuando una gran cantidad de valores existen en un índice, es probable que cada uno de los *bitmaps* sea muy disperso, es decir, muy pocos bits serán 1 en los *bitmaps*, dando por resultado requerimientos de almacenamiento muy grandes para almacenar muchos ceros. En tal caso, se utilizan *bitmaps* comprimidos, o se cambia la representación de bitmap a lista de *RID* como en [O'Neil95].

Usar índices de tipo *bitmap* brinda enormes ventajas en performance y espacio. Para comenzar, el I/O es reducido cuando una fracción grande de una tabla grande es representada usando un *bitmap* y no una lista de *RID*. Por ejemplo, un *bitmap* para un conjunto de 10 millones de registros requerirá un máximo de solamente algunos megabytes de almacenamiento (10 millones de bits = 1.25 millones de bytes) de esta forma los *bitmap* pueden ser almacenados en memoria, y la lista de *RID*s representada se mantiene automáticamente en orden según los *RID*s. Además, las operaciones más comunes usadas para combinar predicados, "Y" y "O", se pueden realizar usando instrucciones muy eficientes que pueden ser ejecutadas en paralelo en 32 o 64 bits en la mayoría de los procesadores.

Por otro lado los *bitmap index* son índices estáticos por lo que en la inserción de un registro todas las entradas del índice tienen que ser desplazadas. [O'Neil97] discute varios tipos de estructuras *bitmap index* convenientes.

El índice *bit-sliced* [Datta99, O'Neil97] sigue el mismo principio que el *projection index* y el *bitmap index*: el valor de la columna indexada para cada registro en la tabla se representa en ese orden en el índice. Sin embargo, cada valor "se rebana" en vez de ser almacenado en su forma original. Es decir, cada posición de bit en la columna que es indexada se almacena en una cadena separada del bloque de datos.

Los *Grid Files* (creados por Nievergelt en 1984) emplea un directorio y una partición del dominio de manera de resolver un pedido con solo dos acceso a disco. El directorio es implementado generalmente por medio de una función de *hash* lo que resulta en grillas de tamaños y formas heterogéneas. El *Excell Index* (Extendible Cell) [Gaede98] (creado por Tamminen en 1992) es una variante del *Grid File* en la que las regiones en las que se descompone el universo es uniforme.

Orientadas a Joins

Puesto que las consultas OLAP requieren generalmente uno o más *joins*, la selección de índices debe considerar los diferentes algoritmos de *joins*. Si bien esta sección no clasifica los índices por su estructura subyacente vale la pena resaltar que, debido al gran volumen de datos y el alto número de joins involucrados, una característica deseable es que los índices permitan resolver rápidamente los joins.

Mientras que los índices tradicionales de tablas mapean los valores de las columnas a los registros que los contiene (generalmente mediante un RID), un *join index* [O'Neil95, O'Neil97] asocia los valores de una columnas de dos tablas. De esta manera, el *join index* representa un join precalculado completamente. Es una forma especial de vista materializada. Organizaciones típicas para un *join index* incluyen *B-trees* o *hash index*. Con *joins* precalculados, también es posible tener acceso a registros de una tabla por valores arbitrarios de una columna en la segunda tabla.

Los *join index* pueden ser generalizados de dos tablas a múltiples tablas. Esta generalización es conocida como *domain index* ya que asocian los valores de un dominio (por ejemplo números de Seguridad Social) a todas las columnas de tablas en la base de datos en las que se encuentren

estos valores. Típicamente, hay solo una tabla para la cual la columna indexada es clave, para las restantes generalmente es una clave externa, y muchos registros pueden contener el mismo valor de columna.

Otra opción a la hora de implementar un índice orientado a *joins*, es el *projection index* [Datta99, O'Neil97]. El mismo consiste en una copia espejo de una columna específica de una tabla. Las entradas en el índice aparecen en el mismo orden que las correspondientes *key-values* de la tabla subyacente, lo que proporciona un mapeo eficiente de las *key-values* a los registros de la tabla. La columna proyectada, que puede o no ser duplicada, es almacenada como una cadena de unos o más bloques de datos.

V. Selección de Vistas e Índices

Los sistemas de Data Warehouse son en general un repositorio de vistas materializadas, diseñadas con un enfoque *global as view*, de datos obtenidos de varias fuentes, integrados y procesados. Pero dado el gran volumen de datos cabe preguntar: ¿Qué vistas hay que materializar? ¿Es posible mejorar el rendimiento del Data Warehouse materializando vistas intermedias? ¿Qué estructuras de datos o índices hay que agregar? Mejoras en la performance no implican solo el tiempo de respuesta a consultas, sino también el tiempo necesario para realizar la actualización de las estructuras subyacentes empleadas.

Los sistemas OLAP están orientados principalmente a operaciones de consultas sobre grandes volúmenes de datos. Los dos principales factores que influyen en el incremento de la performance de este tipo de sistemas son, el almacenamiento de los resultados precalculados, vistas y vistas intermedias, y la utilización de estructuras apropiadas tanto de datos como de índices.

Existen varias estrategias de diseño que apuntan a mejorar la performance de este tipo de sistemas. La trivial y con la que se obtiene el mejor rendimiento es la materialización de todas las vistas, incluyendo las intermedias, junto a varias estructuras de índices. Esto implica diseñar una vista para cada consulta más el conjunto de índices asociado. Esta estrategia requiere mucho espacio de almacenamiento, mucho tiempo de carga del sistema y muchos recursos de administración y gestión de las estructuras de datos e índices. Esto hace que la estrategia tenga un costo implementación demasiado alto para implementarse en la mayoría de los Data Warehouse, mas allá de las ventajas obtenidas en la performance.

El problema que se plantea en la etapa de diseño de un sistema OLAP es el de seleccionar un conjunto de vistas a materializar y un conjunto de índices a crear, que mejore tanto los tiempos de las operaciones de consultas como de actualización, respetando algunas restricciones, principalmente de espacio [Gupta97].

V.1 Costos y Optimizaciones

Al trabajar en un problema de optimización, lo primero a determinar son las medidas o variables sobre las que se debe realizar la misma. En este caso los trabajos de investigación realizados hasta el momento [Gupta97, Harinarayan97, Labio97, Bellatreche00, Ezeife97, Yang97, Harinarayan96] coinciden en que el costo de procesar una consulta Q tiene relación directa con el número de registros procesados por las consultas en los esquemas fuentes, sobre relaciones en las que no se considera la existencia de índices. En caso de existir índices sobre los atributos en los que se realiza las consultas, el costo estaría dado por una constante que dependería del tipo de índice a utilizar. De manera análoga, al verse la actualización de una vista como la ejecución de consultas sobre las bases de datos fuentes, el costo de generación del resultado mantendrá también una relación directa con la cantidad de registros que contenga. En el caso de que las consultas a realizarse contengan sentencias del tipo *GROUP BY*, se debe considerar la cantidad de registros diferentes respecto a los valores en los atributos que se encuentran en la sentencia

[Harinarayan96, Gupta97]. La cantidad puede ser obtenida mediante análisis estadístico de los tipos del atributo considerado.

Otro componente a definir en este tipo de problemas es la función de optimización. Esta se define como la mejora introducida por la materialización de la vista menos el costo de actualización de la misma [Yang97]. De esta manera, si C_{maint} es el costo asociado al mantenimiento y B_Q el beneficio introducido a la consulta Q, el beneficio de materializar la vista en el esquema estará dada por:

$$B_{mat} = B_Q - C_{ma\,\mathrm{int}}$$

B_Q esta dada por la reducción de la cantidad de registros que debe procesar la consulta directa o indirectamente. El beneficio puede verse afectado no solamente por las vistas que se consultan directamente sino también por todas aquellas implicadas en el proceso. De esta manera, si una vista materializada no reduce la cantidad de registros con respecto a las vistas fuentes no debe ser considerada, ya que lo único que agrega es un costo, en este caso de mantenimiento, y no un beneficio. Otro factor implicado es el espacio utilizado tanto para materializar la vista como los índices [Harinarayan97, Bellatreche00].

Otros trabajos consideran más parámetros para la optimización. Por ejemplo [Ezeife97] considera relevantes la memoria utilizada y el beneficio en el uso de la CPU, ya que estos influyen en el tiempo de respuesta de una consulta y por lo tanto considera que deben ser tomados en cuenta.

V.2 Caracterización del Problema

Al estudiar el problema de la selección de vistas e índices en un Data Warehouse, es interesante analizar a priori cual es el orden de magnitud del problema que se pretende atacar.

Suponiendo que se tienen n relaciones a considerar en el problema, la cantidad de posibles combinaciones que permite determinar las vistas que potencialmente pueden ser materializadas, es 2^n. Cualquier subconjunto de potenciales vistas puede ser la posible solución, por lo que el orden del problema es $O\left(2^{2^n}\right)$ [Labio97]. Realizando un razonamiento análogo para el conjunto de índices se obtiene que el orden del problema también es $O\left(2^{2^n}\right)$.

En [Gupta97] se muestra el problema de la selección de índices como una reducción del problema de cubrimiento mínimo de un grafo[4]. Este problema es NP-Completo [Aho83]. Por lo tanto, no existe un algoritmo de orden polinómico que solucione el problema. Por esta razón generalmente las propuestas no buscan la solución óptima sino una solución que sea cercana a esta. Claro está que algunas soluciones se acercan más que otras a la óptima, teniendo como consecuencia que los algoritmos tengan grados de ejecución más cercanos al del problema original. Por lo tanto debe encontrarse un equilibrio entre una solución aceptable y un tiempo de ejecución aceptable para hallar dicha solución.

Los algoritmos que se estudiarán pertenecen a la categoría de algoritmos *greedy* (ávido o exhaustivo) [Aho83], de inteligencia artificial o agentes distribuidos.

V.3 Materializando Vistas

La mayoría de los algoritmos utilizados para resolver este problema pertenecen a la categoría de algoritmos ávidos. La diferencia de las soluciones propuestas radica en el modelo con el cuál se representan las interacciones entre las distintas vistas finales e intermedias.

[4] Otro ejemplo de una reducción a este problema es el problema del agente viajero.

La línea de investigación más fuerte en el tema se basa en el estudio de las consultas más frecuentes, que se modelan con algún tipo de grafo de dependencia que luego es estudiado como un problema conocido en el área de grafos. La aplicación de los algoritmos propicios para dichos problemas ayuda a desarrollar una solución final.

Un enfoque en esta línea es la generación de estructuras multidimensionales de dependencia llamados *lattices* [Harinarayan96]. Los *lattices* son una caracterización de las dependencias entre las vistas como se muestra en la Figura 3. Dado un factor de costo asociado a cada una de las distintas potenciales vistas, basado en la cantidad de registros (a mayor cantidad de registros mayor costo de mantenimiento y mayor rendimiento de las consultas que la utilicen), se establece una relación de dependencia entre ellas. La relación de dependencia se basa en el cálculo del beneficio de agregar una vista al conjunto a materializar, lo que se representa como $B(v,S)$. Esta medida no solo depende del beneficio directo en la ejecución de las consultas que se realizan sobre la vista, sino que también del beneficio indirecto de ser utilizada como fuente intermedia por otras vistas.

Sobre el *lattice* se define un algoritmo de tipo ávido que agrega, en cada paso, la vista con mayor beneficio de todas aquellas que aún no pertenecen a la solución. La cantidad de pasos debe ser estimada a partir de las posibilidades de almacenamiento disponibles para realizar las materializaciones, el cálculo de este valor no se especifica en el trabajo estudiado. Por otro lado sí se establece el mínimo óptimo del algoritmo.

En [Harinarayan96] se demuestra que el algoritmo presentado obtiene un beneficio que es, como mínimo, un 63% del beneficio optimo encontrado por el algoritmo exhaustivo. Establece también, dos casos en los que se obtiene una solución óptima. En aquellos problemas en los que el beneficio de materializar una vista es igual para todas las vistas candidatas y en aquellos en el que existe una vista cuyo beneficio es mucho mayor al del resto de las vistas.

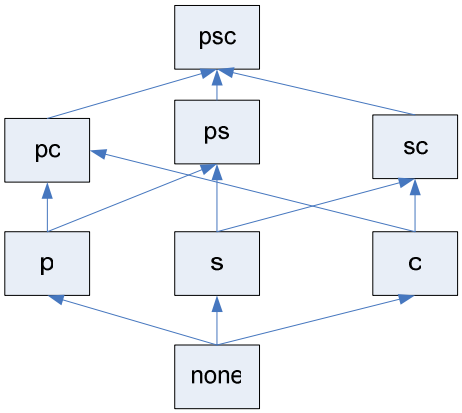

Figura 3: *lattice* para los atributos *part, supplier* y *customer* como se especifica en [TPCD95].

Una adaptación a los *lattices* es mostrada en [Yang97]. Allí se define una estructura llamada *Multiple View Processing Plan* (*MVPP*), un conjunto de árboles cuyos nodos pueden ser comunes. Cada uno de los nodos puede ser, por ejemplo, el *join* de dos tablas o el *join* de resultados anteriores. Cada uno de los árboles se corresponde con el plan de ejecución de las consultas [Elmasri94]. Esto introduce un cambio respecto a [Harinarayan97], ya que en ese trabajo se tomaban como vistas posibles todas las combinaciones de atributos que pudieran darse en el GROUP BY de una consulta. En este trabajo sin embargo se toma en cuenta solo aquellas vistas que pueden ser utilizadas para responder las consultas que el diseñador del Data Warehouse considere pertinente. Por lo tanto, el problema de materialización de vistas se divide en dos etapas: un primer paso que consiste en la generación del *MVPP* a partir de las consultas relevantes y en un segundo que consiste en la selección de las vistas a partir del *MVPP*.

El beneficio se calcula de manera directa como se definió en la sección "*Costos y Optimizaciones*", el costo de mantenimiento de las posibles vistas también está dado por la cantidad de registros que estas posean. La diferencia entre el beneficio de materializar la consulta y el costo de mantenimiento se le llama "peso" de la vista. Generalmente se ordenan las vistas según su peso de mayor a menor, tomando las distintas vistas y recalculando el beneficio total tomando en cuenta las vistas que, según el algoritmo, son parte de la solución. Si la vista mejora al rendimiento total, esta es agregada a la solución. Esto, junto con un algoritmo de generación de *MVPP* conforma el método propuesto.

Si bien este algoritmo es de orden lineal, no se indica que tan efectivo es respecto a la solución óptima. Otro punto que la propuesta no toma en cuenta es el espacio a utilizarse. Por lo tanto se puede seleccionar la materialización de una vista que dé un buen resultado (el mejor según el algoritmo) en cuanto a performance pero no en cuanto espacio consumido, obteniendo así una solución que no sea optima respecto a ambos parámetros.

En [Gupta97] se establecen las posibles vistas a materializarse de manera similar a la de [Harinarayan97], sin embargo las dependencias se definen partiendo de las consultas indicadas por el diseñador disminuyendo así la cantidad de vistas a ser consideradas. El modelo de costo es igual al de [Yang97], solo que se pondera con el tamaño que ocupa la vista. De esta manera, el espacio de disco no solo sirve para deducir cuando se debe detener el algoritmo ávido, sino que se utiliza como parámetro para la selección de la próxima vista a materializar. Aquella vista que tenga un mejor beneficio por unidad de espacio utilizado es agregada a la solución.

Una vez obtenido el conjunto de vistas solución, se pueden ejecutar otros algoritmos, también ávidos, que tomen en cuenta solamente el beneficio y no el espacio de manera de intercambiar vistas. La solución obtenida por este método es al menos tan buena como el obtenido mediante la aplicación del primero. Globalmente se asegura que el beneficio dado por los resultados de [Yang97] es al menos un 63% del obtenido con resultados óptimos.

V.4 Agregando Índices

VISS son las siglas en ingles de "*View, Index and Space Selection*". En este caso el problema de selección de vistas se extiende agregando la selección de índices a crear para el conjunto de vistas elegidas.

La primera estrategia de resolución de este problema consiste en seleccionar los índices luego de haber aplicado uno de los algoritmos vistos en la sección anterior para el cálculo del conjunto de vistas a materializar. El inconveniente que presenta esta estrategia es que puede descartarse una vista que indexada correctamente provea una mejora en el rendimiento, superior que la dada por otra vista que no puede ser indexada. Esto muestra como los algoritmos deben considerar las vistas y los índices de manera simultánea. [Harinarayan97] en su algoritmo *n-greedy* emplea esta técnica. Con un modelo de costo similar al utilizado por [Harinarayan96] se define un algoritmo de tipo ávido en el que en cada paso se toma, o bien una vista con su conjunto de índices o bien un índice de una vista previamente seleccionada. Esta búsqueda se realiza de manera exhaustiva en cada uno de los pasos.

En cada iteración, para el conjunto de elementos seleccionado, en sus variantes de índice o vista-índices, se calcula el beneficio teniendo en cuenta el espacio ocupado por la materialización de estos. Con este algoritmo se mejora sensiblemente el tiempo de ejecución de la selección de la solución. Se tiene un algoritmo de orden km^r, donde k es la cantidad de estructuras en el *lattice,* tal como se especificaba en [Harinarayan96], m es la cantidad de índices y vistas que se eligen para materializar y r tiene relación directa con el orden de ejecución y por lo tanto con la optimalidad de la solución. Esto tiene relación directa con la cantidad de iteraciones que el algoritmo lleve a cabo. Según [Harinarayan97], el beneficio alcanzado por el algoritmo *r-greedy* es al menos $1-1/e^{(r-1)/r}$ veces el alcanzado por la solución óptima del problema.

En el mismo trabajo se introduce también una mejora al algoritmo visto anteriormente llamada *Inner-Level algorithm*, algoritmo tratado en [Gupta97]. La diferencia radica en que los conjuntos que se tienen en cuenta para la materialización no son elegidos de todas las maneras posibles, sino que se seleccionan empleando algoritmos ávidos, seleccionando primero una vista y luego sus índices de forma de obtener el mejor beneficio por unidad de espacio a utilizar. Este algoritmo tiene un mejor orden que el *r-greedy* ($O(k^2m^2)$), además llega a la cota inferior de 0.63 respecto al beneficio obtenido por la solución óptima del problema.

Fuera de la categoría de algoritmos ávidos se pueden encontrar varias propuestas para el tratamiento de este problema. En particular, [Labio97] describe la utilización de un algoritmo A^* utilizado en el campo de la inteligencia artificial [Nilsson71]. El algoritmo se basa en tomar todas las posibles estructuras que puedan estar en el resultado final y revisar el costo. Es muy similar a un algoritmo como los vistos anteriormente, solo que en esta ocasión se recuerdan aquellos casos ya considerados y no tomados en cuenta. Estas vistas e índices no son consideradas en posteriores iteraciones. La evaluación en estos casos se realiza mediante el costo de mantenimiento y beneficio de una manera similar a la que se utiliza en la solución propuesta en [Yang97]. La cantidad de evaluaciones realizadas por este algoritmo puede ser hasta un 99% menor que la realizada por los algoritmos ávidos. Uno de los aportes más importantes de este trabajo es una serie de reglas deducidas a partir de reiteradas ejecuciones del algoritmo sobre distintos entornos característicos. Esto define una serie de heurísticas que pueden ser utilizadas sin necesidad de ejecutar el algoritmo.

Por último, [Bellatreche00] establece un método en el cuál se definen 2 agentes. Uno de ellos, llamado *view spy* tiene el cometido de apropiarse del espacio disponible y dedicarlo a materializar vistas. El otro, llamado *index spy* se encarga de apropiarse del espacio para utilizarlo en la materialización de índices. Entre ambos agentes se pueden quitar espacio de almacenamiento siempre y cuando el *SAC (steal admisión control)*, lo permita. Para ello se emplea un algoritmo basado en un modelo de costos que evalúa si para la solución global es bueno o no. Otro agente, llamado *spy controller,* asegura que solo uno de los dos espías se ejecute a la vez. Este proceso se repite hasta que el *SAC* considere que ya no se pueden realizar más pasajes de espacio de un espía a otro. Los componentes y sus interacciones se pueden ver en la figura 4.

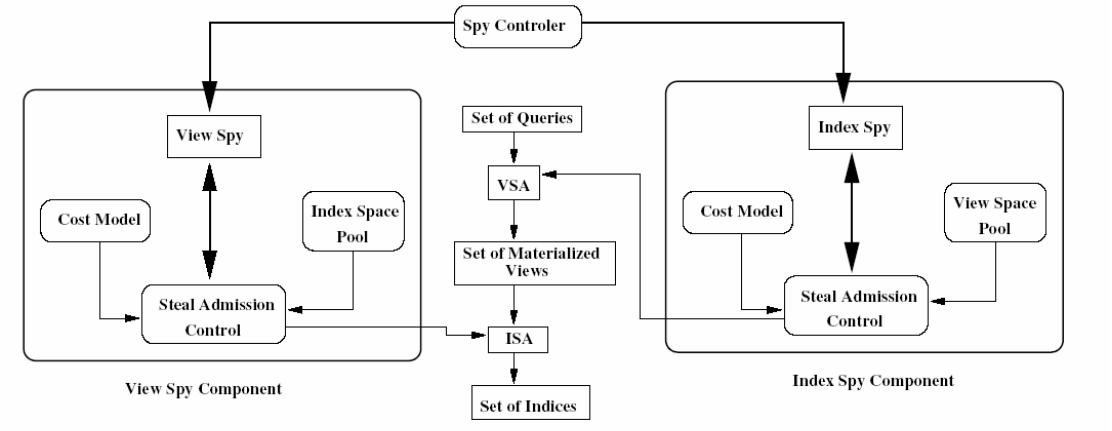

Figura 4: Componentes de la solución basada en agentes

Lo interesante de este enfoque es que el criterio de selección de las vistas e índices, así como el modelo de costo utilizado por el SAC, no se encuentran especificados y podrían, en principio, ser cualquiera de los vistos hasta el momento. Esto convierte la solución en una especie de framework, donde distintos algoritmos podrían ser utilizados.

VI. Caché

Aquellas consultas que a la hora del diseño se suponen serán las más frecuentes y de cálculo más costoso pueden ser materializadas, de manera que no sea necesario recalcularlas cada vez.

La materialización de vistas mejora notoriamente el desempeño de los sistemas de Data Warehouse. La dificultad está en determinar la frecuencia y el costo de las consultas. A priori, el diseñador puede suponer o estimar la frecuencia de ciertas consultas. Sin embargo estos datos pueden cambiar a lo largo de la vida del Data Warehouse. Lo ideal sería realizar la estimación del costo y frecuencia a medida que las consultas se realicen y el sistema evolucione. De esta forma surgió la idea de utilizar caché de resultados en sistemas de Data Warehouse.

En el caché se almacenan los resultados de las consultas para que estas sean utilizadas por otras consultas. Un ejemplo sería el de un usuario que realiza una consulta por las ventas en las distintas tiendas de una determinada ciudad y que luego de estudiar estas medidas, trate de comparar las ventas en la misma ciudad contra las ventas del resto de las ciudades en una zona. En este caso se necesita el resultado de la consulta anterior. Empleando el caché, solo es necesario realizar una sumarización sobre los datos, conocido como *roll-up*, y no recalcular la consulta a partir del cubo. De esta manera, se puede ver el *VIS Problem* como un problema estático de selección de vistas a materializar en el caché.

Según [Scheurmann96] existe una serie de características en los sistemas de Data Warehouse que hacen especialmente atractivo el uso de caché:

- Poco cambio en la información de origen: un mismo conjunto de datos en caché tendrá validez mientras no se realice el mantenimiento del Data Warehouse.
- Tamaño de los resultados respecto a las relaciones origen: generalmente los resultados suelen ser pequeños, como por ejemplo el listado de ventas para una serie de tiendas en una ciudad. Sin embargo, para llegar a estos resultados, los cálculos pueden implicar muchos registros y ciclos de procesador.
- Existencia de patrón de ocurrencia de consultas: típicamente se realiza un *drill-up* o un *drill-down* sobre una consulta ya realizada, por ejemplo.

Las distintas estrategias de implementación de caché puede clasificarse según la granularidad de los elementos a almacenar. Una primera aproximación a esa granularidad esta dada por el trabajo de [Scheurmann96]. En este trabajo el criterio por el cual se decide si una vista puede o no ser almacenada en el caché está dado por la valoración de un resultado que es directamente proporcional al costo de obtención de la vista e inversamente proporcional al tamaño de la misma. Es decir, puede ser preferible materializar el resultado de una consulta costosa de pequeño tamaño que una respuesta de mayor tamaño a una consulta que tiene un costo menor de recalculo. Esta relación se pondera además por su frecuencia de ocurrencia, la cuál es calculada tomando una ventana de *K* consultas del historial de consultas realizadas. El trabajo calcula únicamente la frecuencia para consultas idénticas; no obstante es importante investigar la frecuencia de ocurrencia en el caso de que las consultas sean equivalentes, es decir, devuelven el mismo resultado.

Los sistemas de caché se apoyan en dos algoritmos que se deben implementar: la funcionalidad de agrega al caché y la funcionalidad de reemplazo. La primera determina cuando un resultado debe ser agregado al caché habiendo espacio en el mismo, la segunda determina, dado un caché que está ocupado, cuales son los resultados actualmente en caché que se deben reemplazar y cuando debe hacerse. En este caso se aprovecha el hecho de tener la frecuencia de ejecución de la consulta. De esta manera, en el reemplazo el algoritmo *Least Normalized Cost Replacement* calcula el beneficio como

$$profit(RS_i) = \frac{\lambda_i . c_i}{s_i}$$

siendo λ la frecuencia de ocurrencia de la consulta considerada, *s* una referencia al espacio ocupado por la solución y *c* el costo de la misma (el cuál puede ser calculado en cualquiera de los modelos de costos vistos para el problema *VISS*). En el caso de *Least Normalized Cost Aggregation* se tiene lo que se conoce con el nombre de *e-profit, estimated profit*, y que se calcula como

$$e - profit(RS_i) = \frac{c_i}{s_i}$$

En el caso de la operación de reemplazo se ordena en forma descendente el contenido del caché según el beneficio. Si se deben sacar *n* resultados, siempre los de menor beneficio, para poder agregar el resultado de la consulta actual, entonces el beneficio que se pierde al eliminar estos elementos del caché debe ser menor al ganado por agregar este nuevo resultado. De esta manera, se trata siempre de maximizar el *cost saving ratio*[5]. Para el caso de la operación de agregación se define un mínimo beneficio. Si el beneficio obtenido por agregar RS_i al caché sobrepasa ese valor, entonces se agrega.

La mayor dificultad al implementar el esquema *Watchman* (nombre dado al administrador de caché desarrollado en [Scheurmann96]) es el mantenimiento de los indicadores de frecuencia de ocurrencias de consultas. Un problema típico es que la ocurrencia de una consulta no sea considerada una vez que el resultado de esta ha sido materializado en el caché. Esto repercute en los indicadores de frecuencia, haciendo que la misma sea excluida del caché. Una vez fuera vuelve a ser considerada y posiblemente reingresada al caché. Este problema se conoce como *retained reference information problem* ([O'Neil93]) y debe ser estudiado siempre en estos algoritmos ya que de otra forma se podría estar insertando y reemplazando permanentemente un mismo resultado.

[Deshpande98] busca almacenar elementos de menor granularidad en el caché. En este trabajo la unidad de almacenamiento en el caché es llamada *chunk*. Un *chunk* es una agrupación lógica de datos de un cubo según un conjunto de dimensiones. En este caso, se suele hacer un agrupamiento de las dimensiones siguiendo algún patrón semántico, por ejemplo agrupando en un *chunk* las dimensiones pertenecientes a una misma jerarquía. De esta manera, estableciendo *chunks*, las respuestas se pueden dividir en un conjunto finito de agrupaciones. Todas las agrupaciones que son resultado de una consulta serán medidas por su rendimiento. Se almacena todos los *chunks* de una respuesta o ninguno.

A la hora de evaluar una consulta no es necesario que sea igual a las que se encuentran en el caché, sino que alcanza con que algunos de los fragmentos de los resultados almacenados pueda ser utilizado para su ejecución. Así pues, a la hora de ejecutar una consulta, la misma se divide en dos fragmentos: una subconsulta que puede ser ejecutada por medio de los *chunks* almacenados en el caché y otra que puede ser resuelta en base a las relaciones que se encuentran en el Data Warehouse. Para estos últimos casos, se supone la existencia de una implementación del concepto *chunk* a nivel del DBMS subyacente. De otra manera, el caché no ayudaría en nada a estas consultas y solo servirían para el caso en que todos los fragmentos se encuentren almacenados.

Los conceptos de costos y beneficios vistos en [Harinarayan96] son aplicados en este caso. El número de *chunks* óptimo puede ser calculado mediante una heurística. Aquí sucede lo mismo que con la granularidad en su conjunto: muchos *chunks* comprometen los tiempos de ejecución de búsqueda de correspondencia con las consultas, mientras que pocos *chunks* acerca la solución a la de más alta granularidad, es decir, la consulta en su totalidad.

---

[5] El *cost saving ratio* se calcula como $CSR = \frac{\sum_i c_i.h_i}{\sum_i c_i.r_i}$ donde *hi* es el número de veces que la consulta *i* ha sido satisfecha por el contenido del caché y r_i la cantidad total de veces que la consulta se ha realizado.

CubeStar [Albrecht98] brinda una solución más flexible. En esta solución, los fragmentos a ser utilizados para el almacenamiento son de tamaño variable y se calculan al momento de ser almacenados en el caché. El tamaño de los fragmentos se determina teniendo en cuenta la semántica y el Data Warehouse sobre el que funciona y son almacenados en conjunto para una consulta.

El beneficio que se calcula para los fragmentos implica la participación de más de una métrica. En particular se calculan cuatro métricas que pueden tener distintos pesos dentro del algoritmo, según como este se configure. De esta manera se le puede dar mayor o menor importancia al comportamiento del caché según algunos parámetros independientes al contenido del mismo.

VII. Optimización de Consultas

Si bien la materialización de vistas o el uso de sistemas de caché debe ser transparente para los usuarios del Data Warehouse, es posible que deban realizarse cambios para obtener el máximo provecho de la estrategia seleccionada. Esto implica un proceso de cambio de la consulta original conocido como *query rewritting* [Levy95]. En particular, este problema se reduce a la aplicación serial de dos problemas : en primer lugar, estudiar cuales son los resultados de vistas que total o parcialmente pueden ser utilizados; en segundolugar, minimizar la consulta, es decir, eliminar todos aquellos elementos de la nueva *query* que son redundantes, por ejemplo atributos semánticamente equivalentes que se repiten. La combinación de ambos problemas se conoce con el nombre de *query containment*. Cualquiera de estos problemas resultan ser problemas NP-completos (siéndolo también, en consecuencia, la combinación de ambos), si bien el trabajo muestra un algoritmo lineal que obtiene buenos resultados.

Una variante de este método que trabaja sobre el caso general de *query rewritting* es el trabajo descrito en [Dar96]. En él se estudia el caso particular de consultas en las que las vistas, mediante las cuales se busca la solución, son resultados de consultas que utilizan agregaciones. En este trabajo, los autores especifican reglas de utilización de vista en la reescritura de una consulta, así como algoritmos para los distintos casos. Los algoritmos son de altos grados de complejidad, pero son polinomiales.

Muchos de los trabajos vistos anteriormente, por ejemplo [Albrecht98] o [Deshpande98], toman en cuenta los cambios que se deben realizar en las consultas si se pretende utilizar el esquema de materialización o caché. En particular, se maneja una serie de reglas heurísticas que permite determinar la posibilidad de utilizar o no una vista en el procesamiento de una consulta. En particular, en [Deshpande98] se describen las siguientes reglas:

- Nivel de agregación: Los resultados que se encuentran en caché pueden ser del mismo nivel de agregación o de un nivel inferior. De otra manera, no se puede utilizar este resultado para calcular el nuevo resultado a devolver.
- Lista de proyección: Los atributos utilizados en la proyección realizada en la nueva consulta debe ser un subconjunto de los atributos utilizados en la proyección de la vista almacenada en el caché.
- Lista de selección: Los atributos que se encuentran en el *group by* de la consulta deben coincidir exactamente. En caso de que los atributos no se encuentren en el *group by*, no tienen porque realizarse una correspondencia exacta.

El conjunto de reglas especificado en [Dar96] es mayor y más complejo y completo.

Mas allá de que la mayoría de los trabajos en el área de optimización de consultas se enfoquen hacia a la materialización de vistas o bien en el caché de resultados, la aplicación en cualquiera de las dos categorías es indistinta. Esto debido a que cualquier de los dos casos puede verse como una materialización de vistas.

VIII.Fragmentación y Distribución

Los sistemas OLAP poseen generalmente bases de datos de mayor tamaño que los sistemas fuentes OLTP, esto debido a los datos histórico, la des-normalización utilizada en la implementación relacional y el calculo de atributos principalmente del tipo *meassures* [Kimball96].

Dado el gran volumen de información, la fragmentación tanto horizontal como vertical permite obtener respuestas más veloces a las consultas que se realizan sobre el Data Warehouse. Es pertinente saber en que casos es conveniente una distribución de datos entre más de un nodo en una red. Al igual que en las bases de datos distribuidas [Tamer91], se puede pensar en distribuir los distintos fragmentos en varios nodos. Esto permitiría explotar el principio de localidad de datos así como la posibilidad de emplear técnicas de paralelismo.

VIII.1 Fragmentación

Vertical

Cuando en la literatura existente se habla del problema de fragmentación de datos, se refiere no solo a la división de una relación, sino también a la unión de relaciones. Fragmentar verticalmente los datos [Tamer91] es sumamente beneficioso tanto para mantener acotado el tamaño de las relaciones, como para poder paralelizar el acceso a estas.

En los Data Warehouse, esta fragmentación es aún más importante debido a la existencia de una operación que es propia de las aplicaciones del tipo OLAP: los *drill-across* [Abello02]. Estas operaciones involucran consultas de datos de más de un cubo. Estas podrían haber sido materializados a vistas distintas, sin embargo, para estas operaciones la materialización considerada puede no ser la de mejor rendimiento (ninguno de los algoritmos vistos considera este tipo de consultas). Es por ello que el conjunto de vistas, tanto primarias como secundarias, deberían ser revisadas.

[Golfarelli01] describe un algoritmo para la realización de revisiones como las que se describieron. Basándose en un conjunto de vistas materializadas, halladas por algún algoritmo de los existentes (el trabajo se apoya en el trabajo de [Harinarayan96]), este algoritmo define un particionamiento basado en el conjunto de consultas frecuentes. Esta partición no debe quedar solamente en el conjunto de vistas finales a materializar debido a la redundancia de datos que existe en este entorno, sino que es necesario conocer también sobre que tablas se basa cada una de las consultas. Este problema no existe en una partición vertical realizada sobre una base de datos transaccional debido a que la redundancia de datos no existe.

Para calcular las fragmentaciones a realizar, el algoritmo se basa en la generación de una matriz de tres dimensiones: las posibles particiones (vistas estas como un conjunto de atributos), las posibles combinaciones de medidas que se aplican sobre estas particiones y por último las consultas existentes. Una terna con valor 1 indica que la consulta puede ejecutarse sobre la partición y medidas en cuestión, en caso contrario su valor será 0. La creación de una partición tiene asociada un costo que se define como la cantidad de páginas de disco a las que se debe acceder. Esto se calcula teniendo en cuenta la cantidad de relaciones, la cantidad de registros accedida y la cantidad de registros por página de disco. Luego se debe determinar que particiones quedan y cuales no.

Así como el problema de materialización de vistas e índices, este problema también es NP-completo (este problema se deriva del problema de la mochila del tipo 0-1 [Weiss95]). En este caso [Golfarelli01] no propone un algoritmo ávido para la solución del problema, sino que se basa en una técnica de diseño de algoritmos llamada *Branch-and-Bround* [Aho83]. En este enfoque, se colocan las posibles soluciones como hojas de un árbol hasta que estas formen un subárbol con un costo mayor que el mínimo alcanzado hasta el momento. De esta manera, se tiene una cantidad de soluciones a revisar mucho menor que en la búsqueda exhaustiva. En el

peor caso, se puede tener el mismo orden que la búsqueda exhaustiva (O(n!)), pero en el caso promedio el orden es mucho menor.

Horizontal

El trabajo más representativo respecto a la fragmentación horizontal de Data Warehouse en implementaciones relacionales es el realizado por Noaman y Barker [Noaman99]. En este trabajo, la propuesta se centra en el particionamiento horizontal de la tabla de medidas, dejando invariante las relaciones de dimensión. La motivación está dada por la gran diferencia de tamaño entre la relación de medidas y las de dimensiones. De esta manera, las relaciones de dimensión se mantendrán intactas mientras que se fragmenta la de medidas.

La fragmentación se realiza siguiendo el enfoque de *fragmentación horizontal indirecta* [Tamer91] debido a que las restricciones sobre las que se basa el enfoque dependerán de las tablas asociadas, en particular de dimensión. No es útil realizar una fragmentación basada en restricciones sobre los valores de las medidas, ya que, como se verá en la próxima sección, generalmente no tiene sentido.

De esta manera, el trabajo especifica un algoritmo que se basa en tres pasos:

- Optimización de los predicados derivados de las consultas: Los predicados se realizan sobre las dimensiones del cubo y por lo tanto sobre las relaciones de dimensión definidas. La idea es realizar una normalización a nivel de la jerarquía sobre la que se define. Para ello se elije la mayor granularidad.
- Generación de las particiones de las dimensiones: Luego de la generación de *minterms* [Tamer91] (descomposiciones mínimas de los predicados en forma conjuntiva), es posible realizar la fragmentación de las distintas dimensiones, tomando para ello datos de las aplicaciones que se ejecutan sobre las relaciones.
- Generación de las particiones de la relación de medidas: Una vez que se tiene la fragmentación de las dimensiones, se puede deducir cuales son los fragmentos relacionados a la tabla de medidas.

El trabajo realiza también el estudio de la correctitud de los resultados mediante la demostración de que las soluciones cumplirán con las propiedades de completitud, reconstrucción y no solapamiento.

VIII.2 Distribución

Una vez realizadas las fragmentaciones se debe elegir la ubicación de los fragmentos. El enfoque tradicional contempla principalmente el caso de un único repositorio de datos en el cual se tiene la base de datos que implementa el Data Warehouse. La utilización de un Data Warehouse en sitios remotos, sumado a la gran cantidad de datos que las herramientas OLAP manipulan cuestiona los beneficios de esta solución. Principalmente cuando los enlaces que comunican los sitios con el repositorio son de escaso ancho de banda.

[Noaman99] establece la distribución de los fragmentos de la tabla de medidas según las consultas que se ejecuten en los distintos lugares y que dan origen a los *minterms* que sirven de *input* al algoritmo. Sin embargo, no se establece si esta distribución se debe realizar siguiendo un patrón estadístico respecto a la ocurrencia de las consultas en los distintos sitios. Las relaciones de dimensiones se replican en todos los sitios. Esto se basa en la baja frecuencia de renovación de las dimensiones y la necesidad de acceso a los datos en todos los sitios que acceden al Data Warehouse.

[Golfarelli01] no ataca el problema de la distribución de las particiones verticales. No obstante, es natural pensar en un enfoque similar al que se tiene en [Noaman99], sobre todo pensando en que las particiones verticales se desprenden de un algoritmo que utiliza consultas como entradas. En este caso algunas particiones pueden tener que repetirse en más de un sitio dependiendo de la frecuencia de operaciones del tipo *drill-accross*.

Arquitecturas Distribuidas

Independientemente de cómo se distribuyan los fragmentos, se debe establecer una arquitectura que permita la visión transparente del Data Warehouse por parte de cualquier usuario.

La arquitectura que se muestra en la figura 5 se encuentra descripta en [Noaman99].

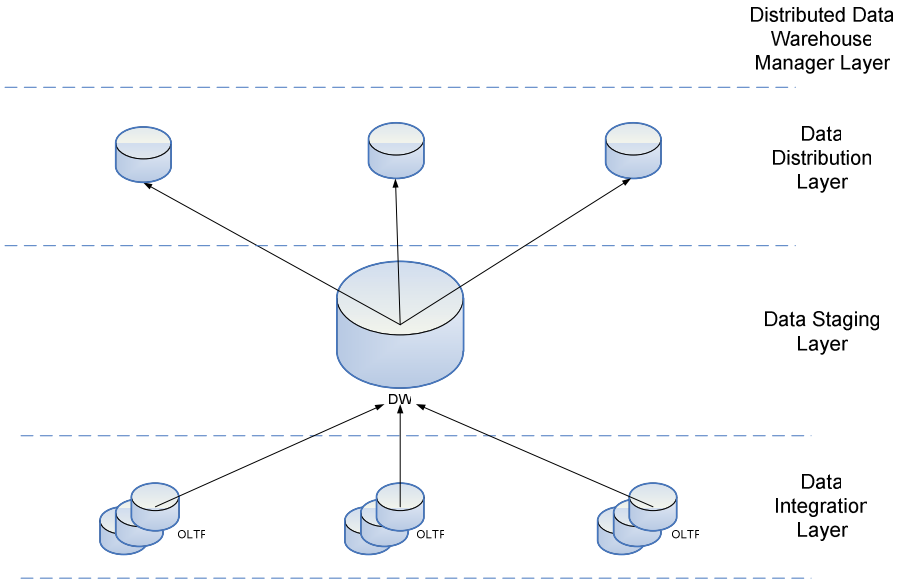

Figura 5: Arquitectura para Data Warehouse distribuído

Esta arquitectura se basa fuertemente en la arquitectura de bases de datos distribuidas vista en [Tamer91], que se basa a su vez en el modelo ANSI/SPARC [Elmasri94]. Cada una de las cuatro capas que la compone cumple una función:

- Data Integration Layer: En esta capa se realiza la integración de datos de distintas fuentes de datos transaccionales (OLTP).
- Data Staging Layer: Esta capa realiza la unificación en un único esquema conocido como *integrated conceptual schema*. Es el esquema que cualquiera de los usuarios del Data Warehouse tendrán disponible para realizar sus consultas.
- Data Distribution Layer: Cumple con las funciones de fragmentación, distribución y mantenimiento de los datos de las bases que contienen el Data Warehouse en los distintos nodos.
- Distributed Data Warehouse Manager Layer: Realiza el manejo de los fragmentos locales y provee una vista única de los fragmentos al usuario final.

Debe contarse con una red de interconexión entre los distintos nodos para la carga, actualización y consulta de datos que no se encuentran en el fragmento local del Data Warehouse.

IX. Carga y Mantenimiento

El proceso de generación de vistas establece también la manera en que estas deben cargarse, en el caso de un Data Warehouse relacional las consultas SQL que definen las vistas. Existen propuestas basadas en el diseño lógico del Data Warehouse como el presentado en [Larrañaga03]. Sin embargo, una vez que los procesos de optimización basados en materialización de vistas e índices son ejecutados, el esquema resultante puede ser totalmente diferente del planificado en el diseño lógico.

Teniendo las sentencias que definen las vistas a materializar, es directa la manera de realizar la carga en el Data Warehouse. Se ejecutan las distintas sentencias para las relaciones de sumarización o vistas intermedias, en el orden establecido por las precedencias establecidas en el algoritmo, por ejemplo a través del *lattice* definido en [Harinarayan96]. Una vez ejecutado el proceso, se tiene el Data Warehouse con los datos materializados en las distintas vistas. Este proceso suele realizarse antes de que el Data Warehouse quede disponible a las consultas realizadas por los usuarios o herramientas OLAP.

Para el mantenimiento de vistas lo más común es que el contenido del Data Warehouse sea totalmente recalculado en cada proceso de carga a partir de los datos de fuente. Dependiendo del tipo de sistema existen variantes como las descriptas en [Chaudhuri97]. De cualquier forma, todas las técnicas se basan en tomar los datos de las fuentes, sean estos la totalidad o la porción de datos nuevos, e insertarlos en las relaciones que implementan el o los cubos multidimensionales. En estos períodos de tiempo, el Data Warehouse se encuentra en estado *off-line*, para que el usuario no pueda acceder a los datos mientras se realiza la actualización y por lo tanto no se tenga una visión inconsistente de los mismos.

Este enfoque puede ser aplicado en bases de datos de pequeño o mediano porte o para la caga inicial. Para Data Warehouses de mayor tamaño o con necesidades de alta disponibilidad, que no permite tiempos *off-line*, esto no es lo más apropiado. De esta forma se abre una nueva rama de investigación en la cuál hay dos enfoques. Por un lado, precalcular las sumarizaciones para luego integrarlas al Data Warehouse, mediante un algoritmo que toma los datos de manera totalmente incremental. Por otro lado, utilizar técnicas de auto-mantenimiento, *self-maintenability* [Gupta95].

IX.1 Carga Inicial

La carga inicial eficiente [Fenkx00, Hjaltason97] es un tema relevante cuando se trata de una cantidad muy grande de datos, en especial sobre una tabla indexada. Al crear un nuevo índice es deseable proporcionarlo tan rápidamente como sea posible a los usuarios y brindar la mejor performance para las consultas. El proceso de iniciación y mantenimiento de un Data Warehouse es un proceso crítico. El gran volumen de datos implicado dicta la necesidad de un proceso de carga de alto rendimiento. La capacidad de almacenar eficientemente conjuntos extensos puede tener un efecto dramático en el costo total asociado al mantenimiento de un Data Warehouses.

Varios trabajos consideran el problema de carga inicial en las estructuras de índice multidimensionales, como [Garcia98] para los *R-Trees*, [Hjaltason97] para *Quad-Trees* y [Fenkx00] para *UB-Trees*.

IX.2 Carga Incremental

El proceso de carga incremental se diferencia del proceso de carga inicial ya que además de crear páginas nuevas de datos, como lo hace el proceso de carga inicial, actualiza las páginas existentes.

Delta Summarization

Supongamos un Data Warehouse en donde se almacena el total de ventas del producto *A* por localidad. Cada venta que se realice del producto *A* en la localidad, se sumará al valor del registro que pertenezca al producto *A* y que pertenezca a la localidad que se estudia. Esto funciona sin que se tenga que saber el total de la tabla. Solamente el nuevo valor de venta y el valor anterior son suficientes para que un nuevo cálculo sea realizado.

En este principio se basa [Mumick97] para realizar la actualización de un Data Warehouse. El proceso de mantenimiento se divide en dos etapas: *propagación* y *actualización*. El trabajo se centra en las relaciones de sumarización, es decir sobre las tablas de medidas, más allá de que se describe también un método para el mantenimiento de las relaciones de dimensión.

En la primera etapa, se realiza el mantenimiento de estructuras auxiliares llamadas *summary-delta tables*. Existe una por cada vista que se encuentra materializada en el Data Warehouse. Las mismas se generan a partir de tres vistas virtuales definidas por cada una de las relaciones que se encuentran en las bases fuente. En ellas se guardan los cambios realizados y una medida que mantiene la relación respecto a la sumarización realizada en la relación destino. De esta manera se tiene la vista *prepare-insertions* para las inserciones y *prepare-deletions* para el borrado en las relaciones de origen. Una tercera vista realiza la consolidación de ambas. Por ejemplo, si en la relación destino se tiene como medida la suma de los valores X (*SUM(X)*), entonces se guarda X. En la siguiente tabla se pueden apreciar los mapeos entre las funciones de agregación de las vistas destino y las que guardan las vistas virtuales.

	Prepare-insertions	Prepare-deletions
COUNT(*)	1	-1
COUNT(exp.)	Cuando exp. Es null➔ 0, sino 1	Cuando exp. Es null➔ 0, sino -1
SUM(exp.)	*exp.*	*exp.*
MIN(exp.)	*exp.*	*exp.*
MAX(exp.)	*exp.*	*exp.*

Tabla 1: Mapeos de las funciones de agregación en [Mumick97]

Agregando los datos almacenados en las vistas *prepare-changes,* se tiene las *summary-delta tables*. Es importante observar, y este es el mayor aporte de este enfoque, que el trabajo de calculo ya queda realizado en gran parte en esta etapa de propagación. Todo el trabajo hasta este momento se realizó sin tener la necesidad de colocar *off-line* el Data Warehouse, ya que los datos de las vistas materializadas no han sido utilizados.

Recién en la segunda etapa (*actualización*) los datos del Data Warehouse son modificados. Estos datos son calculados en base a los datos ya existentes en la sumarización y a los datos que se encuentran en su *summary-delta table*. Por ejemplo, si el dato que se guarda en el Data Warehouse es un SUM(X), basta con sumarle el valor que se encuentra en su tabla auxiliar. Cada registro que se encuentre en las *summary-delta tables*, implica recalcular el valor de a lo sumo un registro en la relación destino. Si alguno de los cruzamientos no se encuentra en el Data Warehouse, volviendo al ejemplo anterior podría ser un producto que es vendido por primera vez en una localidad, se inserta un nuevo registro. Si el valor de la sumarización pasa a ser 0, debido a que se borraron algunos de los registros en las relaciones origen, entonces puede optarse por eliminar el registro del Data Warehouse.

Este enfoque funciona tanto para inserciones como para borrados en tablas orígenes. Los únicos casos en los cuales no es aplicable, lo que implica que el recalculo del valor debe ser realizado a partir de todas la relación fuente, es cuando se trabaja con eliminaciones y la sumarización es un *MIN* o un *MAX*. Estas funciones no son *self-manteinable*, es decir, el nuevo valor se puede calcular únicamente con el cambio y con el valor anterior, respecto a los borrados. Por ejemplo, si el valor mínimo (máximo) calculado hasta ahora es seis y se elimina un registro cuya propiedad tiene el valor seis, no es posible determinar si seis sigue siendo el mínimo (máximo) o no. Esto determina un caso en los cuales las *summary-delta tables* no tienen mayor utilidad y el cálculo debe realizarse desde las relaciones de origen.

En el caso de que exista más de una vista materializada, tanto auxiliares como finales, se puede utilizar la relación de orden dada por [Harinarayan96] para realizar ordenadamente el proceso de propagación. Esto hace aún más útil tanto la implementación de vistas auxiliares como la utilización de *summary-delta tables* para el mantenimiento de las tablas. La dificultad para aplicar este método es el hecho de materializar las *summary-delta tables*. Esto implica duplicar el espacio necesario para el DBMS. Más allá de que el espacio de las vistas se utilice mientras se desarrollen las etapas del algoritmo, este debe estar disponible y puede hacer que falle todo el proceso en caso de que no se cumpla el requisito.

Las mejoras están dadas por el hecho de que los cálculos más pesados del mantenimiento se realizan con el Data Warehouse operable y la etapa de *actualización* en la cuál el Data Warehouse no estaría disponible es tan larga como la cantidad de registros que se hayan cambiado. Un enfoque en donde esto se conjugue con un buen aprovechamiento del espacio determinaría un esquema altamente aceptable.

Auto-mantenimiento

Todas las propuestas que han sido tratadas hasta el momento se basan en disponer de las relaciones fuentes a la hora de realizar el mantenimiento de las tablas. Para los casos en los que no se puede o no se quiere acceder a las relaciones fuentes, por ejemplo, por tener arquitecturas distribuidas, una alternativa para el mantenimiento de vistas materializadas se llama *self-maintenance* [Gupta95]. En este trabajo se define el término para aquellas vistas que pueden pasar de un estado a otro, sin basarse en sus relaciones fuentes: solamente se toma en cuenta el estado anterior de la vista y el *update* que se realizó sobre la relación fuente.

No todas las vistas son auto-mantenibles. Es por eso que [Huyn97] no solamente basa su algoritmo en esta propiedad, sino que establece también un mecanismo por el cual se determina si una vista que ha sido materializada es o no "auto-mantenible". Este se basa en propiedades de 2 consultas llamadas DIFF e INCON. Estas consultas, derivadas de la consultas cuya aplicación sobre las relaciones fuentes retorna la vista materializada, pueden aplicarse a un algoritmo que determina mediante la relación ≤ definida en [Harinarayan96][6], en tiempo polinómico si DIFF≤INCON. En ese caso, se dice que la vista que se examina es "auto-mantenible".

El trabajo establece además que independientemente de las relaciones que dieron origen a la vista "auto-mantenible", los cambios hechos sobre esas relaciones van a llevar la vista de una situación estable a otra situación estable. En estas posibles relaciones, se encuentran las que definen la base canónica de un conjunto de vistas. Esta base se puede deducir de las consultas que definen las vistas y de la vista materializada, *sin necesidad de conocer la fuente*. Por lo tanto, una vez que se demostró que una vista es auto-mantenible, basta con hallar la base canónica del conjunto de vistas que se quiere mantener. De esta manera, aplicando los cambios que se realizan a las bases de datos fuentes también a la base de datos canónica, se puede deducir un nuevo estado, que va a ser consistente con el estado posterior de las vistas materializadas. Basta entonces con llevar esos cambios de la base canónica a las vistas del Data Warehouse.

Cabe la posibilidad de que una vista no sea *self-manteinable*, sino que necesite de ciertas relaciones de la base de datos fuente para que se pueda realizar su mantenimiento. A este tipo de vistas se le llama auto-mantenibles generalizadas. La solución, en un Data Warehouse, para estos casos pasa por materializar las relaciones que hacen la vista auto-mantenible. De esta manera se consigue que el mantenimiento del Data Warehouse se pueda realizar sin tener que ejecutar operaciones sobre las base de datos fuente.

Este enfoque tiene como desventaja lo que para el enfoque [Mumick97] era una ventaja: el Data Warehouse tiene que ser puesto en estado *off-line* mientras se llevan a cabo estas operaciones. A esto debe sumarse que se deben implementar algunos mecanismos para que el motor de bases de datos del Data Warehouse pueda recibir las consultas de los DBMS fuentes. Sin embargo, el mantenimiento sería menor que el que se debía realizar en el caso original. Aún más, la disponibilidad de las tablas, como puede suceder en el caso de las arquitecturas distribuidas, no afecta el rendimiento ni el resultado y por lo tanto el Data Warehouse siempre tendrá las actualizaciones que el administrador/diseñador crea oportuno.

[6] La relación ≤ se define en [Harinarayan96] de la siguiente manera: Sea *a* y *b* los resultados de dos consultas q_a y q_b, se dice que $q_a \leq q_b$ sii *a* es un subconjunto de *b*.

X. Discusión

Ningún elemento de diseño[7] de los descriptos son por si solos, capaces de cumplir con todos los propósitos de un buen diseño físico. Por esta razón, generalmente, se utilizan en conjunto

En general los trabajos continúan la línea propuesta por otros, como por ejemplo las estrategias propuestas en [Golfarelli01, Harinarayan96, Harinarayan97], lo que permite definir una estrategia global de diseño. En términos generales, se podría identificar la siguiente secuencia de pasos al diseñar físicamente un Data Warehouse:

- Seleccionar el **Modelo de Datos**
- Seleccionar una **Estrategia de Compresión**
- Seleccionar las **Estructuras de Índices**
- Aplicar una **Estrategia de Resolución del "VIS Problem"** para la selección y materialización de las vistas e índices. Complementariamente se puede optar por una **Estrategia de Caché**
- Seleccionar un **Método de Optimización de Consultas**
- Seleccionar los **Procesos de Carga y Mantenimiento**

Si bien en general este es el orden de pasos seguido, existen variantes basadas en las restricciones impuestas al diseñador. Un ejemplo de estas restricciones es que el tiempo en que el Data Warehouse se encuentre off-line esté acotado. Esto condiciona los procesos de mantenimiento, que influyen a su vez en las estructuras de datos y espacio consumido. Otras restricciones comunes vienen dadas por el tipo de uso del Data Warehouse. Estos ejemplos confirman la existencia de relaciones entre los distintos elementos de diseño, que en algunos casos se complementan y en otros casos son excluyentes. Algunos de estos aspectos son discutidos en esta sección.

X.1 ROLAP vs. MOLAP

La elección de un sistema ROLAP o MOLAP depende en gran medida de la infraestructura disponible al momento de implementar el OLAP. Mucho de los proveedores de bases de datos relacionales (por ejemplo ORACLE o IBM) tienen módulos que permiten expandir su manejador relacional con funcionalidades de análisis multidimensional. Esto hace que las soluciones ROLAP sean efectivas desde el punto de vista de los costos.

Sin embargo, las características de las bases de datos relacionales (en cuanto a su almacenamiento y los lenguajes de manipulación existentes) hacen que las operaciones OLAP no tengan buen desempeño. Las implementaciones MOLAP son más efectivas desde el punto de vista de las velocidades de respuesta. Como contraparte, sin el uso de un método de compresión de datos, el espacio a utilizarse puede ser importante.

En general optar por una estratega del tipo MOLAP suele ser la mejor alternativa, si bien muchas veces los recursos (en particular monetario) no lo permiten, teniendo que optar por una estrategia del tipo ROLAP.

X.2 Selección de Estructuras de Índices

Al igual que en los sistemas OLTP, la elección de las estructura de índice a emplear depende de varios factores, en particular de los tipos de datos a ser utilizados y del coeficiente costo beneficio (ya que suele existir una penalización grande en espacio).

[7] Se entiende por elemento de diseño estrategia de selección de modelo de datos y de índices, algoritmo de optimización del "VIS Problem", entre otros nombrados en el presente trabajo.

El tipo de datos funciona como un factor excluyente a la hora de utilizar algunos tipos de índices. Por ejemplo, todos los índices que se basan en estructuras de árboles pueden ser aplicados sobre tipos de datos ordenados. Para tipo de datos no ordenado se debe definir una función de orden, lo que implica un costo extra. Otros tipos de índices aceptan cualquier tipo de datos. Sin embargo el desempeño de estos suele ser menor, debido a la ejecución de la función de hash necesaria para catalogar un dato dado.

Otro punto importante es el costo del mantenimiento de los índices. Este punto debe tomarse en cuenta sobre todo en aquellos ambientes en los cuales la carga del Data Warehouse debe hacerse en franjas horarias acotadas. Nuevamente se está ante un problema de mejora global de la performance del sistema. Debe obtenerse un óptimo a la ejecución de consultas y al mantenimiento del sistema.

Finalmente, el espacio utilizado es importante como parámetro de diseño de Data Warehouses. Esta característica del índice elegido impacta, por ejemplo, a la hora de resolver el problema VISS.

La tabla 2 presenta una comparación entre las principales características de los distintos índices presentados en la sección IV. Entre las características resaltables, se encuentran los costos (tanto de mantenimiento, como de espacio ocupado), tipos de dato sobre los cuales se aplican (por ejemplo si es necesario que los datos puedan ser ordenados o no), estructura sobre las cuales pueden ser aplicados (MOLAP o ROLAP) y cuál es el ambiente de uso más apropiado.

	Tipo	Uso	Estructura	Tipo de Datos	Uso de Espacio	Costo de Mantenimiento
B^+-Tree	Arboles	Tabla de dimensiones	ROLAP-MOLAP	Ordenado	Alto (varios niveles de referenciación).	Alto. Se debe reestructurar ante datos insertados
UB-Tree	Arboles	Valores que pueden ser divididos en rangos	ROLAP-MOLAP	Ordenado	Medio (almacena únicamente los límites de los rangos)	Medio - Alto (un nuevo valor insertado no necesariamente implica reestructuración. En caso de límite de sección la reestructuración es grande)
R-Tree	Arboles	Valores que pueden ser divididos en rangos	ROLAP-MOLAP	Ordenado	Medio (almacena únicamente los límites de los rangos)	Medio - Alto (un nuevo valor insertado no necesariamente implica reestructuración. En caso de límite de sección la reestructuración es grande)
X-Tree	Arboles	Cantidad grandes de dimensiones a indexar	ROLAP-MOLAP	Ordenado	Medio (almacena únicamente los límites de los rangos)	Medio - Alto (un nuevo valor insertado no necesariamente implica reestructuración. En caso de límite de sección la reestructuración es grande)
DC-Tree	Arboles	Valores que pueden ser divididos en rangos	ROLAP-MOLAP	Ordenado	Medio (almacena únicamente los límites de los rangos)	Medio - Alto (un nuevo valor insertado no necesariamente implica reestructuración. En caso de límite de sección la reestructuración es grande)
Linear Hashing	Hash	Tabla de Dimensiones	ROLAP	Cualquiera	Bajo (Solo se necesita un arreglo de buckets y punteros)	Medio (se debe hallar el bucket correspondiente al puntero y actualizarlo o agregarlo)
Budy Tree	Hash	Valores que pueden ser divididos en rangos	ROLAP-MOLAP	Ordenado	Medio (almacena únicamente los límites de los rangos)	Medio - Alto (un nuevo valor insertado no necesariamente implica reestructuración. En caso de límite de sección la reestructuración es grande)
Bitmap	Hash	Tabla de Dimensiones	ROLAP-MOLAP	Cualquiera	Alto (varios niveles de referenciación y cada hoja contiene un arrgelo de bits).	Medio - Alto (se debe modificar el arreglo de bits. En caso de que el dato no existiera puede haber reestructuración)
Bit-sliced	Hash	Tabla de dimensiones	ROLAP-MOLAP	Ordenado	Medio (almacena únicamente los límites de los *slices*)	Medio (Si cambia el dato puede estar dentro de un slice (no hay cambio) o en un límite (se cambia dicho límite)
k-d-tree	Hash - Arboles	Tabla de dimensiones	ROLAP-MOLAP	Ordenado	Alto (existen muchas dereferenciaciones hasta llegar al dato)	Bajo (el árbol no se encuentra balanceado)
Quad-Tree	Hash-Arbol	Tabla de dimensiones	MOLAP	Ordenado	Alto (utiliza cubos auxiliares para el almacenamiento de los datos)	Bajo (Cambios en los cubos auxiliares o en el límite registrado en el subcubo)
Excell	Hash-Arbol	Valores que pueden ser divididos en rangos	ROLAP-MOLAP	Ordenado	Medio (almacena únicamente los límites de los rangos)	Medio - Alto (puede haber una reestructuración debido a que las celdas son de igual tamaño).

Tabla 2:Principales características de los indices vistos en la sección IV

X.3 Caché vs. Materialización de Vistas

Se debe realizar un análisis del uso del Data Warehouse, de las características de este y del tipo de consultas a resolver para decidir cuál es la estrategia más adecuada.

La materialización es más eficiente a la hora de realizar consultas, ya que las vistas se materializan al cargar el Data Warehouse, pero como el conjunto de vistas materializadas es estático es necesario materializar todas las vistas usadas frecuentemente junto a sus índices. Esto puede insumir mucho espacio en disco.

Por otro lado el empleo de caché reduce el espacio necesario debido a que el conjunto de vistas materializadas es dinámico. Sin embargo, la ejecución de las operaciones de administración insume tiempo al igual que la creación de las vistas que no se encuentran en el caché.

Los costos de manejo del administrador de caché se ven compensados por la mejora de rendimiento de las consultas. [Deshpande98] muestra resultados de experimentos donde se obtiene mejoras de un 60% en los tiempos de ejecución de consultas. Mientras el esquema de materialización de vistas descrito en [Harinarayan96] muestra mejoras que a lo sumo llegan a un 40%. Estos resultados se basan en experiencias donde se emplearon frecuencias no uniformes de consultas, por lo que es de esperar que los rendimientos de los enfoques de caché y materialización no sean tan dispares en un entrono normal de uso.

XI. Conclusiones

Este trabajo presenta un estado del arte del diseño físico de sistemas OLAP, el que requiere técnicas completamente distintas al de los sistemas OLTP. Dado el gran volumen de información involucrado y las operaciones a las que está orientado un sistema de este tipo. Existe un gran número de trabajos realizados en esta área, si bien la mayoría tratan el tema en el área relacional, en particular proponen nuevas estructuras de índices y nuevos algoritmos para la resolución del "VIS Problem". Los trabajos realizados sobre diseño físico *MOLAP* se basan principalmente en estructuras de datos, mientras que los realizados sobre *ROLAP* lo hacen en la adaptación de mecanismos existentes para bases de datos relacionales. Esto por ser MOLAP un paradigma nuevo y específico, mientras que ROLAP se basa en el modelo relacional.

En general el diseño eficiente de un sistema no puede involucrar etapas desconexas que resuelvan parte del problema desconociendo los requerimientos de los subsiguientes pasos. El diseño físico de Data Warehouse no es una excepción por lo que debe ser tratado como un proceso global en donde los distintos componentes, que hacen al diseño físico, se conjuguen de manera que se puedan cumplir con los objetivos para los cuales el Data Warehouse está siendo creado. Si bien varios trabajos se basan en otros previos no se encontraron trabajos sobre metodologías globales de diseño.

Tres de los puntos que centran la atención de las investigaciones en esta área son, la resolución del "VIS Problem", estructuras eficientes de índices y el de mantenimiento del Data Warehouse. Esto debido al gran volumen de datos involucrado, requerirse tiempos cuasi-interactivos en al momento de trabajar con sistemas de este tipo y requerirse cortos plazos en los que el sistema esté off-line.

Otro aspecto a destacar es el bajo acoplamiento existente entre los trabajos realizados sobre Diseño Físico y Diseño Lógico de un Data Warehouse. Algunos aspectos han sido tratados, como por ejemplo la carga y mantenimiento. Sin embargo, no se han encontrado trabajos que permitan seleccionar, por ejemplo, la estrategia de diseño físico del lógico (relación clara a la hora de definir una metodología para la solución del *VISS*) o viceversa. Sería interesante, por lo tanto, lograr un nexo entre estos dos niveles de diseño, de manera que ambas áreas se enriquezcan con los avances de las otras.

Referencias

[Abello02] A. Abelló, J. Samos, F. Saltor: "*On Relationships Offering New Drill-Across Possibilities*", DOLAP' 02, Nov. 2002.

[Agrawal95] R. Agrawal, A. Gupta, S. Sarawagi: *"Modeling Multidimensional Databases."*, Technical Report, IBM Almaden Research Center, 1995.

[Aho83] A. Aho, J. Hopcroft, J. Ullman: "*Estructuras de Datos y Algoritmos*", Addison-Wesley Iberoamericana, Ene. 1983.

[Albrecht98] J. Albrecht, A. Bauer, O. Deyerling, H. Günzel, W. Hümmer, W. Lehner, L. Schlesinger: "*Management Of Multidimensional Aggregates For Efficient Online Analytical Processing*", Proceedings of the 1999 International Symposium on Database Engineering & Applications, Feb. 1999.

[ARB] Arbor Software Corporation, Sunnyvale, CA. Multidimensional Analysis: Converting Corporate Data into Strategic Information. http://www.arborsoft.com/.

[Barbara97] D. Barbara, M. Sullivan: *"Quasi-Cubes: A Space Efficient Way To Support Approximate Multidimensional Databases."*, Technical Report, ISE Dept., September 1997.

[Bayer97] R. Bayer, V. Markl: *"The UB-Tree: Performance Of Multidimensional Range Queries."*, Technical Report TUMI9814, Institut f¨ur Informatik, TU Mûnchen, 1997.

[Bellatreche00] L. Bellatreche, K. Karlapalem, M. Scheneider: "*On Efficient Storage Space Distribution Among Materialized Views and Indices in Data Warehousing Environments*", Proceedings of the ninth international conference on Information and knowledge management, Mar. 2000.

[Berchtold96] S. Berchtold, D. A. Keim, H.P. Kriegel: *"The X-Tree: An Index Structure for High-Dimensional Data."*, Proc. 22th Int. Conf. on Very Large Data Bases, Bombay, India, 1996, pp. 28-39.

[BOS] Business Objetcs, Business Objetcs SA, http://www.businessobjects.com

[CGN] Cognos, Cognos Inc., http://www.cognos.com/

[Chaudhuri97] S. Chaudhuri, U. Dayal: "*An Overview Of Data Warehousing And OLAP Technology*", ACM SIGMOD Record 26(1), Mar. 1997.

[Dar96] S. Dar, H. V. Jagadish, A. Y. Levy, and D. Srivastava: "*Answering SQL Queries With Aggregation Using Views.*", In Proc. of VLDB, pages 318-329, Set. 1996.

[Datta99] A. Datta, K. Ramamritham, H. Thomas: *"Curio: A Novel Solution For Efficient Storage And Indexing In Data Warehouses."*, Proceedings on International Conference on VLDVm, Septiembre 1999.

[DB2] DB2 Warehouse, IBM Inc., http://www-306.ibm.com/software/data/db2/datawarehouse/

[Deshpande98] P. Deshpande, K. Ramasamy, A. Shulka, J. Naughton: "*Caching Multidimensional Queries Using Chunks*", 27th. International Conference on the Management of Data (SIGMOD '98), Seattle, USA, Jun. 1998.

[Elmasri94] R. Elmasri, S. Navathe: "*Fundamentals Of Database Systems*", Second Edition, Benjamin/Cummings, 1994.

[Ester01] M. Ester, J. Kohlhammer, H. Kriegel: *"The DC-Tree: A Fully Dynamic Index Structure for Data Warehouses."*, Proc. 16th Int. Conf. on Data Engineering (ICDE 2000), 2000

[Ezeife97] C.I. Ezeife: "*A Uniform Approach For Selecting Views And Indexes In A Data Warehouse*",, Proceedings of the 1997 International Database Engineering and Applications Symposium (IDEAS), Abr. 1997.

[Fenkx00] R. Fenkx, A. Kawakami, V. Marklx, R. Bayerx, S. Osaki: *"Bulk loading A Data Warehouse Built Upon A UB-Tree."*, IDEAS Conf. 2000, Yokohama, Japan,2000

[Furtado00] P. Furtado, H. Madeira: *"FCompress: A New Technique For Querible Compression Of Facts And Datacubes."*, International Database Engineering and Applications Symposium, IDEAS'2000,Sep. 2000

[Gaede98] V. Gaede, O. Günther: *"Multidimensional Access Methods."*, ACM Computing Surveys, Vol. 30, No. 2, 1998, pp. 170-231.

[Garcia98] Y. J. García, M. A. Lopez, S. T. Leutenegger: *"A Greedy Algorithm For Bulk Loading R-Trees."*, In ACM International Workshop on Advances in Geographic Information Systems, pages 163–164, 1998.

[Goldstein98] J. Goldstein, R. Ramakrishnan, U. Shaft: *"Compressing Relations And Indexes."*, Proceedings of the 14th. Int. Conf. On Data Engineering, Febrero 1998, Florida, EE.UU..

[Golfarelli01] R. Goffarelli, D. Maio, S Ritzzi: "*Appliying Vertical Fragmentation Techniques In Logical Design Of Multidimensional Databases*", Very Large Database (VLDB) conference, May. 2001.

[Golfarelli01b] M. Golfarelli, S. Rizzi, B. Vrdoljak: *"Data Warehouse Design From XML Sources"*, Proceedings of the 4th ACM international workshop on Data warehousing and OLAP, November 2001.

[Gray96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh: *"Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals."*, Proceedings of the 12th International Conference on Data Engineering, pages 152-159, 1996.

[Gupta95] A. Gupta, I. Mumick: "*Maintenance of Materialized Views: Problems, Technics And Application*", IEEE Data Engineering Bulletin, Special Issue on Materialized View & Data Warehousing, Jun. 1995.

[Gupta97] H. Gupta: "*Selection Of Views To Materialize In A Data Warehouse*", 6th. International Conference on database Theory (ICDT 97), Mar. 1997.

[Harinarayan96] V. Harinarayan, A. Rajaraman, J. Ullman: "*Implementing Data Cubes Efficiently*", In ACM SIGMOD International Conference on .Management of Data, Jun. 1996.

[Harinarayan97] H. Gupta, V. Harinarayan, A. Rajaraman, J. Ullman: "*Index Selection For OLAP*", In International Conference on Data Engineering, Burmingham, U.K, 1997.

[Hjaltason97] G. R. Hjaltason, H. Samet, Y. J. Sussmann: *"Speeding Up Bulk-Loading Of Quadtrees"*, In ACM International Workshop on Advances in Geographic Information Systems, pages 50–53, 1997.

[Hümmer03] W. Hümmer, A. Bauer, G. Harde: *"XML And Architecture: Xcube: XML For Data Warehouses"*, Proceedings of the 6th ACM international workshop on Data warehousing and OLAP, 2003.

[Huyn97] N. Huyn: "*Multiple-View Self-Maintenance In Data Warehousing Environments*", Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, Greece, Mar. 1997.

[IRI] IRI Software, Information Resources Inc., Waltham, MA. OLAP: Turning Corporate Data into Business Intelligence. http://www.infores.com/.

[ISF] O3 Software, Ideasoft, http://www.ideasoft.com.uy/

[Kimball96] R. Kimball: "*The Data Warehouse Toolkit*",, John Wiley, 1996.

[Labio97] W. Labio, D, Quass, B. Adelberg: "*Physical Database Design For Data Warehouses*", Proceedings of the International Conference on Data Engineering (ICDT 97), Jun. 1997.

[Larrañaga03] I. Larrañaga: "*Carga Y Actualización De Data Warehouses A Partir Del Diseño Lógico De Su Esquema*", Reporte Interno del grupo CSi,InCo, 2003

[Levy95] A. Levy, A. 0. Mendelson, Y. Sagiv, and D. Srivastava: "*Answering Queries Using Views.*", In Proc. ACM PODS, pages 95-104, Mar. 1995.

[MIC] Microstrategy Inc., Vienna, VA 22182. True Relational OLAP. http://www.microstrategy.com.

[MSS] Microsoft Analysis Services, SQL Server, Microsoft Corporation. http://www.microsoft.com/sql/analysisservicesdefault.asp

[Mumick97] I. S. Mumick, D. Quass, B. S. Mumick: "*Maintenance Of Data Cubes And Summary Tables In A Warehouse*", In Proceedings of the ACM SIGMOD Intemational Conference on Management of Data, May. 1997.

[Nguyen01] T.B. Nguyen, A.M. Tjoa, O. Mangisengi: "*MetaCube-X: An XML Metadata Foundation For Interoperability Search Among Web Warehouses.*", In: DMDW, Interlaken, Switzerland, 2001

[Nilsson71] N. Nilson: "*Problem Solving In Artificial Intelligence*", McGraw-Hill, 1971.

[Noaman99] A.Y. Noaman, K. Barker: "*Distributed Data Warehouse Architecture And Design*", ISCIS 99, Kusadasi, Turquia., Jun. 1999.

[O'Neil93] E. O'Neil, P. O'Neil, G. Weikum: "*The LRU-K Page Replacement Algorithm For Database Disk Buffering*", Proceedings of the ACM SIGMOD International Conference on Management of Data, Oct. 1993.

[O'Neil95] P. O'Neil, G. Graefe: "*Multi-Table Joins Through Bitmapped Join Indices*", SIGMOD Record 24(3), 1995, pp. 8-11.

[O'Neil97] P. O'Neil, D. Quass: "*Improved Query Performance With Variant Indices*", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 38-49.

[ORA] Warehouse Builder, Oracle Inc., http://www.oracle.com/technology/products/warehouse/index.html

[RED] http://www-306.ibm.com/software/data/informix/redbrick/

[Roussopoulos97] N. Roussopoulos, Y. Kotidis, M. Roussopoulos: "*Cubetree: Organization Of And Bulk Incremental Updates On The Data Cube*", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 89-99.

[Scheurmann96] P. Scheurmann, J. Shim, and R. Vingralek: "*WATCHMAN: A Data Warehouse Intelligent Cache Manager.*", In Proc. of VLDB, pages 51-62, Feb. 1996.

[Srivastava02] J. Li, J. Srivastava: "*Efficient Aggregation Algorithms For Compressed Data Warehouses.*", IEEE Trans. Knowl. Data Eng. 14,2002

[Tamer91] M. Tamer, P. Valduriez: "*Principles Of Distributed Database Systems*",, Prentice Hall, 1991.

[TPCD95] F. Raab, editor: "*TPC Benchmark D (Desicion Support)*", Revision 1.0, Transactiojn Processing Performance Council, Abr. 1995.

[Vitter99] J.S. Vitter, M. Wang: "*Approximate Computation Of Multidimensional Aggregates Of Sparse Data Using Wavelets.*", 1999 Conf. On the Management of Data, Philadelphia, PA, USA, 1999.

[Weiss95] M.A. Weiss: "*Estructuras De Datos Y Algoritmos*",, Adisson-Wesley Iberoamericana., Jul. 1995.

[Yang97] J. Yang, K. Karlapalem, Q. Li: "*Algorithms For Materialized View Design In Data Warehousing Environment*", Proceedings of the International Conference Very Large Databases (VLDB), Ago. 1997.

[Zhuge95] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom: "*View Maintenance In A Warehousing Environment*", Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 316-327, May 1995.