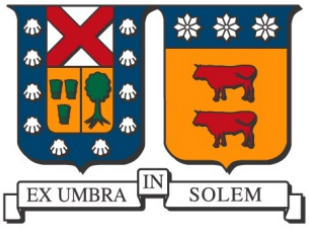
2019-07

ESTUDIO E IMPLEMENTACIÓN DE UN MÉTODO DE COMPRESIÓN DE DOCUMENTOS PARA BÚSQUEDAS EFICIENTES EN MEMORIA DE RECURSOS DEL OBSERVATORIO VIRTUAL

SALDÍAS GONZÁLEZ, CAMILO ALEJANDRO

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“ESTUDIO E IMPLEMENTACIÓN DE UN MÉTODO DE COMPRESIÓN DE DOCUMENTOS PARA BÚSQUEDAS EFICIENTES EN MEMORIA DE RECURSOS DEL OBSERVATORIO VIRTUAL”

CAMILO ALEJANDRO SALDÍAS GONZÁLEZ

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Diego Arroyuelo B.
Profesor Correferente: Mauricio Araya L.

Julio - 2019

DEDICATORIA

Este trabajo está dedicado a mis padres, quienes han sido mi más grande apoyo en todos los aspectos de mi vida. No sería lo que soy, y no hubiese llegado tan lejos, si no hubiese sido por su amor y apoyo incondicional.

AGRADECIMIENTOS

Este trabajo no hubiera sido posible sin el apoyo de muchas personas, tanto profesional como personalmente.

En primera instancia me gustaría agradecer a mi profesor guía, Diego Arroyuelo. Su pasión y entrega a su disciplina son inspiradoras, su atención a los detalles es digna de admirar, y su apoyo durante todo este proceso fue fundamental para completar este trabajo. Trabajar con usted ha sido todo un placer, y le agradezco enormemente todo el apoyo que me ha brindado a lo largo de mi carrera.

También me gustaría agradecer al proyecto del Observatorio Virtual Chileno, en especial al profesor Mauricio Araya y a Camilo Nuñez, quienes tuvieron la paciencia de responder todas mis preguntas y proveer recursos de cómputo para la realización de este trabajo.

Además me gustaría agradecer al equipo de FabLab UTFSM, con el cual compartí gran parte del tiempo que utilicé para desarrollar este trabajo, además de gran parte de mis últimos años de Universidad. Participar en este proyecto me enseñó muchísimas cosas que no se aprenden en la sala de clases, además de permitir desarrollarme personal y profesionalmente, y la posibilidad de contribuir al desarrollo de mi Universidad.

No puedo dejar pasar la oportunidad de agradecer a mis amigos, quienes han estado conmigo apoyándome en las buenas y en las malas. En especial a Alex, Felipe, Camila, Carlos, Sebastián, Pablo, Ignacio, Joaquín, y a todos aquellos que me tendieron una mano cuando más lo necesité. Son bakanes, muchas gracias por formar parte de mi vida.

Por último, me gustaría agradecer a mi familia por brindarme todo el amor y apoyo que necesité a lo largo de mi paso por la Universidad. Siempre me brindaron el apoyo para que pudiera estudiar lo que quisiera, aún cuando ello implicara cambiarme de carrera, pero por sobre todo me brindaron los medios para que pudiera estudiar y desarrollarme sin preocuparme de nada más. Los amo, muchas gracias por todo.

RESUMEN

Resumen— En este trabajo se presenta una solución que permite la compresión de documentos provenientes de recursos del Observatorio Virtual, que además posibilita la consulta eficiente de los mismos en memoria. Lo anterior se logra mediante el uso de estructuras sucintas que otorgan tradeoffs entre espacio comprimido y tiempo de consulta. El desempeño de múltiples estructuras sucintas logran resultados interesantes en tiempo y en espacio en el contexto del Observatorio Virtual, siendo estos resultados en muchos casos superiores en tiempo y espacio a lo obtenido con un popular motor de bases de datos relacional sobre los mismos datos. Además, se muestra una aplicación concreta, implementando un prototipo de sistema de búsqueda basado en la solución planteada en este trabajo. Estos resultados muestran que es posible lograr espacios y tiempos competitivos aplicando estructuras sucintas al problema de indexación y búsqueda de recursos del Observatorio Virtual, posibilitando el desarrollo de este tipo de sistemas que irán en directo beneficio de la comunidad científica y astronómica nacional.

Palabras Clave— Estructuras Sucintas; Observatorio Virtual; XML; Obtención de Atributos.

ABSTRACT

***Abstract*—** In this work, a solution is presented that allows the compression of documents obtained from resources of the Virtual Observatory, while enabling efficient querying of them in memory. This is achieved by using succint data structures that offer tradeoffs between the space required to store the data and time required to query it. We show that the performance of multiple succinct structures that achieve interesting results in time and space in the context of the Virtual Observatory, where these results are in many cases superior in time and space to what was obtained with a popular relational database engine over the same data. In addition, a real-world application is shown, implementing a prototype search system based on the solution proposed in this work. These results show that it is possible to achieve competitive space and time results by applying succinct structures to the problem of indexing and searching resources of the Virtual Observatory, enabling the development of this type of systems that will directly benefit the national astronomical and scientific community.

***Keywords*—** Succint Structures; Virtual Observatory; XML; Attribute Retrieval.

GLOSARIO

ADQL: Astronomical Data Query Language.
ALMA: Atacama Large Millimeter/submillimeter Array.
API: Application Programming Interface.
BPS: Balanced Parentheses Support.
BWT: Burrows–Wheeler Transform.
CSA: Compressed Suffix Array.
CST: Compressed Suffix Tree.
ChiVO: Chilean Virtual Observatory.
DAP: Data Access Protocol.
DaCHS: GAVO Data Center Helper Suite.
ESA: European Space Agency.
ESO: European Southern Observatory.
FITS: Flexible Image Transport System.
GaVO: German Astrophysical Virtual Observatory.
HTTP: Hypertext Transfer Protocol.
ISA: Inverse Suffix Array.
IVOA: International Virtual Observatory Alliance.
LCP: Longest Commom Prefix Array.
NAOJ: National Astronomical Observatory of Japan.
OV: Observatorio Virtual.
SA: Suffix Array.
SCS: Simple Cone Search.
SDSL: Succint Data Structure Library.
SIAP: Simple Image Access Protocol.
SQL: Structured Query Language.
SSAP: Simple Spectra Access Protocol.
TAP: Table Access Protocol.
UCD: Unified Content Descriptor.
UTFSM: Universidad Técnica Federico Santa María.
WT: Wavelet Tree.
XML: eXtensible Markup Language.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

El Observatorio Virtual (OV) es un proyecto con múltiples centros de datos e instituciones a nivel mundial, los cuales proveen a los astrónomos de múltiples datasets y servicios complementarios, comúnmente especializados en ciertas áreas de la astronomía. Toda esta información se encuentra disponible en forma de múltiples servicios estandarizados, los que pueden ser accedidos por medio de aplicaciones y librerías presentes en múltiples lenguajes de programación, lo que hace sencillo el acceso y uso de estos recursos. Sin embargo, el proceso de descubrimiento y búsqueda de estos datos y recursos no es algo sencillo, ya que requiere conocimiento técnico por parte del usuario para acceder a estos servicios, además de conocer qué servicios proveen los datos que el usuario necesita. En este sentido, se hace necesaria la creación de un sistema que permita a un usuario, mediante una consulta, descubrir los servicios que contienen los datos que le son relevantes, simplificando así la tarea de conocer la ubicación de esa información en el Observatorio Virtual y su posterior acceso.

Este trabajo se enfoca en el estudio e implementación de un método que permita comprimir los documentos que describen servicios del Observatorio Virtual en memoria, permitiendo realizar búsquedas sobre ellos en memoria de forma eficiente. Esto, con el objetivo de posibilitar la construcción de sistemas que permitan la búsqueda y descubrimiento de datos a través de múltiples servicios y ubicaciones en el Observatorio Virtual. Para realizar lo anterior, se discutirán y probarán múltiples alternativas de estructuras de datos sucintas y algoritmos para manejar las consultas realizadas a este sistema, buscando obtener el mejor balance entre tiempo de respuesta y espacio necesario para mantener esta información.

En primera instancia se realiza una breve introducción al problema que se está abordando, además de especificar los objetivos de este trabajo. Posteriormente se establecerá el marco conceptual de este trabajo, incluyendo la estructura y arquitectura del Observatorio Virtual y las distintas estructuras de datos sucintas que existen, además de mencionar aspectos prácticos de la implementación de estas últimas a través de la librería SDSL. Luego se procede a presentar la propuesta de solución, la que abarca desde la determinación de las fuentes de datos, pasando por la determinación de la estructura sucinta a utilizar, hasta la metodología para procesar las consultas y una arquitectura para un prototipo basado en esta solución propuesta. A continuación se procede a validar esta propuesta, presentando resultados de experimentos realizados sobre datos extraídos del Observatorio Virtual, finalizando con conclusiones acerca de lo expuesto a lo largo de este trabajo.

El código utilizado para realizar los experimentos presentados en este trabajo, así como el prototipo desarrollado, se encuentran en el siguiente repositorio de GitHub:

`https://github.com/csaldias/memoria`

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

La *International Virtual Observatory Alliance* (IVOA) es una organización internacional fundada en Junio del 2002, con la misión de facilitar la coordinación y colaboración internacional para el desarrollo e implementación de las herramientas, sistemas y estructuras organizacionales necesarias para posibilitar el libre uso de datos astronómicos como un observatorio virtual integrado e interoperable[1]. En la actualidad, la IVOA está compuesta de más de 20 programas de Observatorio Virtual (OV) repartidos en los 5 continentes[2], cada uno de los cuales ofrece múltiples servicios de datos a la comunidad astronómica y científica global. Cada uno de los servicios proveídos por los múltiples programas de OV sigue un estándar definido por la IVOA, de forma de que el acceso a los datos pueda ser realizado por cualquier astrónomo, independiente de la fuente o servicio utilizado para la obtención de los mismos.

Dada la diversidad geográfica con la que cuenta el OV a nivel mundial, es de esperarse que cada programa de OV cuente con un set de características únicas que los diferencien de otros programas de OV, como los tipos de datos que manejan, sus fuentes o instrumentos específicos utilizados para su obtención, o la porción de cielo o del espectro radioeléctrico en la que se realizan estas observaciones. En la actualidad la obtención de esta información se realiza a través de un proceso manual, en donde es necesario obtener directamente las características de cada servicio provisto por cada programa de OV de forma separada. A pesar de que existen registros centralizados que contienen las características de los distintos OV alrededor del mundo, además de formas de acceder a este contenido, no existe un sistema que permita realizar búsquedas semánticas sobre los servicios o sobre las tablas contenidos en los mismos, de forma de realizar búsquedas exploratorias para encontrar dónde se encuentra la información más relevante para el usuario. Más específicamente, en ChiVO se reconoce la necesidad de contar con un servicio de estas características, de manera de simplificar la labor de buscar el servicio correcto para el tipo de dato, observación o instrumento requerido, entre otras posibles características.

Es por lo anteriormente mencionado que en este trabajo se plantea el estudio de métodos de compresión de documentos que permitan mantener una base de datos de documentos (en este caso, características de los servicios antes mencionados) que posibilite su búsqueda y almacenamiento en memoria de forma eficiente, junto a la implementación de un prototipo que permita consultas a esta base de datos desde otras aplicaciones a través de una API REST.

[1] IVOA. "What is the IVOA". http://www.ivoa.net/about/what-is-ivoa.html. Consultado el 26 de Octubre de 2018.

[2] IVOA. "Member Organizations". http://www.ivoa.net/about/member-organizations.html. Consultado el 26 de Octubre de 2018.

1.1. Objetivos

El objetivo general de este trabajo consiste en estudiar e implementar un método de compresión de los documentos que contienen la información de los recursos del OV, tal que ocupe el menor espacio posible en memoria y permita la rápida consulta de los documentos en cuestión.

1.1.1. Objetivos Específicos

Para poder cumplir con el objetivo general previamente propuesto es necesario completar los siguientes objetivos específicos:

- Analizar los diferentes métodos de compresión de documentos presentes en la literatura, haciendo énfasis en la rapidez de consulta de los documentos comprimidos.
- Realizar un estudio experimental de los documentos que contienen la información de los recursos del OV, para determinar el mejor método de compresión para esta aplicación en particular.
- Implementar un prototipo de compresión de estos documentos, que además permita realizar consultas a los mismos a través de una API REST por parte de otras aplicaciones del OV.

CAPÍTULO 2
MARCO CONCEPTUAL

A continuación, se procederá a describir el marco conceptual que actuará como fundamento teórico para el método de compresión de documentos a elegir e implementar.

2.1. Arquitectura y Datos de un OV

2.1.1. Arquitectura de un OV

El proyecto *Virtual Observatory* (o VO por sus siglas en inglés) permite a los astrónomos consultar múltiples centros de datos de forma sencilla y transparente, entregando nuevas herramientas de análisis y visualización dentro de ese sistema, y entrega a los centros de datos un *framework* estándar para la publicación y entrega de servicios utilizando sus datos, el cual está compuesto por un conjunto de estándares y protocolos elaborados y publicados por la IVOA.

Este *framework* o arquitectura[3] estándar, definida por la IVOA para asegurar la interoperabilidad de los distintos proyectos de OV a nivel mundial, está compuesta de 3 capas, apreciables en la Figura 1:

- *Resource Layer*, donde se encuentra la colección de datos provistos por el OV.
- *User Layer*, donde los usuarios del OV acceden a los datos de la capa de recursos a través de diferentes servicios.
- *Middle Layer*, que permite la conexión entre ambas capas antes mencionadas, escondiendo la complejidad al usuario final.

Además de las 3 capas antes mencionadas, la IVOA define un marco técnico para que los OV compartan sus datos y servicios, a través de un *Registry* y una serie de *Data Access Protocols*. El *Registry* [Benson *et al.*, 2009] funciona como el directorio del OV, recolectando metadatos acerca de los datos y servicios de información en una base de datos consultable. Tal como los recursos y servicios del OV, el *Registry* también es distribuido, existiendo diferentes réplicas tanto para redundancia como para albergar colecciones de datos más especializadas.

Por otro lado, el acceso a los datos y a la colección de metadatos se realiza a través de una serie de *Data Access Protocols* (DAP), los cuales especifican una manera estandarizada de

[3] http://www.ivoa.net/documents/Notes/IVOAArchitecture/

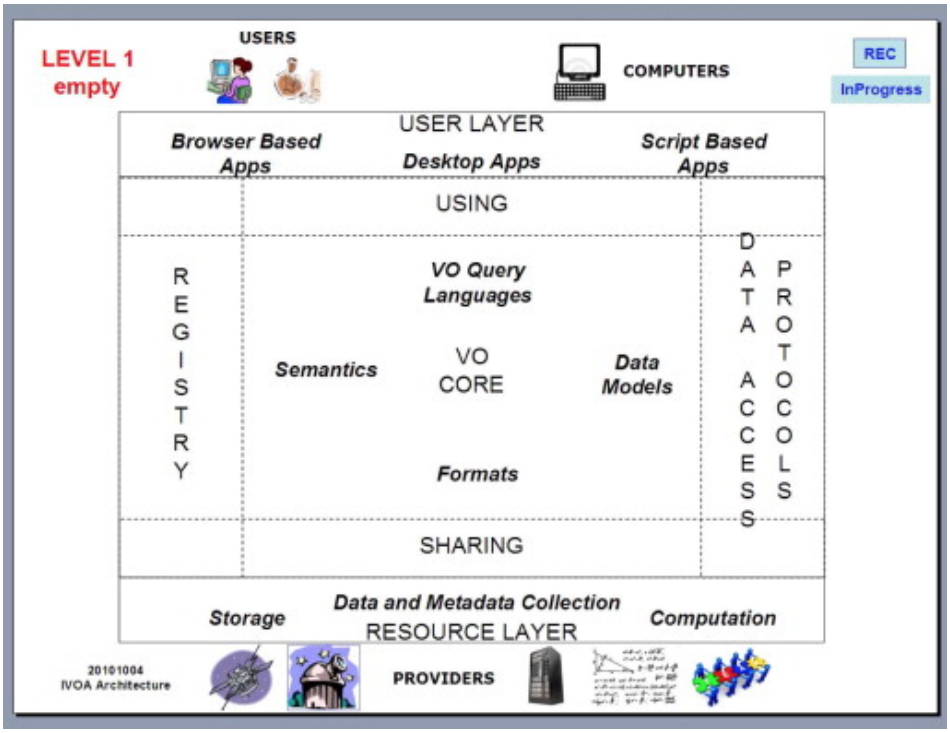


Figura 1: Nivel 1 de la Arquitectura IVOA (abstracción de alto nivel).
Fuente: [Araya *et al.*, 2015].

obtener datos y metadatos desde diferentes proveedores. En la actualidad, los servicios de DAP más utilizados [Araya *et al.*, 2015] son:

- *Simple Cone Search* [Williams *et al.*, 2008], o SCS, que permite consultar por una porción del cielo (especificada en coordenadas astronómicas) sobre la cual buscar por recursos.
- *Table Acess Protocol* [Dowler *et al.*, 2010], o TAP, un servicio que permite acceder a información tabular, que provee acceso a datos tabulares y a metadatos de tablas y bases de datos.
- *Simple Image Access Protocol* [Dowler *et al.*, 2015], o SIAP, un servicio que permite la obtención de imágenes de una zona del cielo.
- *Simple Spectra Access Protocol* [Tody *et al.*, 2012], o SSAP, un servicio que permite acceder a espectros unidimensionales provenientes de colecciones de datos y metadatos.

En este trabajo, nos enfocaremos principalmente a datos provenientes del *Table Access Protocol*. Además de lo anterior, es importante considerar los siguientes estándares relacionados con el manejo y obtención de los datos del OV:

- *Astronomical Data Query Language* [Ortiz *et al.*, 2006], o ADQL, un lenguaje de consulta fuertemente basado en SQL92 que incluye operaciones específicas de astronomía.
- *Virtual Observatory Table* [Ochsenbein *et al.*, 2011], o VOTable, un estándar XML para el intercambio de datos (y metadatos) astronómicos en forma de tablas, basado en el estándar FITS de tablas astronómicas.

2.1.2. Tipos de Archivos

Tal como se mencionó anteriormente, en un OV se manejan principalmente datos y metadatos de observaciones astronómicas. Dado que se trabajará con datos provenientes del OV, es importante conocer los distintos tipos de archivos que se encuentran en el OV.

En el caso de datos astronómicos, el formato de archivo más popular es FITS [Hanisch *et al.*, 2001], o *Flexible Image Transport System*, un formato originalmente desarrollado a fines de la década de los 70 para permitir el intercambio de datos visuales astronómicos entre diferentes sistemas y plataformas computacionales, con diferentes tamaños de símbolos y diferentes formas de expresar valores numéricos. Este formato es mucho más que un simple formato de imagen[4], como lo son JPEG o GIF, y es usado para el transporte, análisis y archivado de *datasets* astronómicos, tales como arreglos multidimensionales, tablas conteniendo múltiples filas y columnas de información, y *keywords* que proveen información descriptiva acerca de los datos contenidos.

Los servicios de DAP entregan la información referente a los servicios del OV en el formato VOTable [Ochsenbein *et al.*, 2011], un estándar XML para el intercambio de datos representado como un conjunto de tablas, con un énfasis particular en tablas astronómicas y basado en el estándar FITS para tablas. Su estructura fundamental se muestra en la Tabla 1. Junto a la VOTable, se utiliza un vocabulario de keyworkds estandarizado para la descripción de los campos de la VOTable, llamado *Unified Content Descriptors*, o UCD. Este vocabulario formal para datos astronómicos es controlado por la IVOA para evitar la mayor cantidad de ambigüedades posibles, y es restringido para evitar la proliferación de términos y sinónimos dentro del vocabulario.

2.1.3. ChiVO, el Observatorio Virtual Chileno

El proyecto nacional de Observatorio Virtual es ChiVO [Solar *et al.*, 2015], proyecto fundado el año 2013 con el objetivo inicial de disponer de forma pública de los datos del Observatorio ALMA a la comunidad científica chilena y global. La creación de ChiVO busca satisfacer las siguientes necesidades de la comunidad astronómica nacional:

[4] "What is FITS?" https://fits.gsfc.nasa.gov/, consultado el 16 de Diciembre de 2018.

Tabla 1: Estructura de una VOTable
Fuente: Elaboración Propia, basado en [Ochsenbein *et al.*, 2011].

VOTable	**Metadatos** + **Datos** asociados, organizados en múltiples **Tablas**
Metadatos	**Parámetros** + **Información** + **Descripciones** + **Links** + **Campos** + **Grupos**
Tabla	lista de **Campos** + **Datos**
Datos	conjunto de **Filas**
Filas	conjunto de **Celdas**
Celda	**Primitiva**, o lista variable de **Primitivas**, o arreglo multidimensional de **Primitivas**

- **Descubrir** datos astronómicos de un objeto o instrumento en una porción determinada del cielo o espectro radioeléctrico, ya sea por búsqueda o por exploración.
- **Obtener** los datos requeridos en distintos formatos, ya sea desde el VO o desde un servicio externo.
- **Comparar** la información de datos obtenidos desde diferentes fuentes de información.

Para la implementación de la arquitectura y servicios especificados por la IVOA, ChiVO utiliza la suite DaCHS [Demleitner *et al.*, 2014], creada por el Observatorio Virtual Alemán GAVO. Esta suite incluye componentes para la ingesta y mapeo de datos, implementaciones de la mayoría de los servicios de búsqueda y acceso a datos, y los protocolos de acceso a datos especificados por la IVOA, los cuales cubren la mayoría de las necesidades de la comunidad astronómica chilena [Solar *et al.*, 2015]. La arquitectura resultante de esta implementación se muestra en la Figura 2, en la cual se puede apreciar que para el caso de los recursos de ALMA, se han implementado los 4 servicios de DAP más utilizados mencionados con anterioridad.

En [Solar *et al.*, 2015] se menciona que las consultas hacia ChiVO se realizan a través de consultas en HTTP, GET o POST, y la lista de resultados es retornada por el *Endpoint* en una tabla XML con formato VOTable. Dado que en esta memoria se trabajará principalmente con los metadatos asociados a los datos presentes en ChiVO, nuestro foco principal será el procesamiento de los archivos VOTable que describen los recursos disponibles en el OV.

2.2. Compresión de documentos

El problema de compresión de documentos es un problema ampliamente abordado en la literatura [Reghbati, 1981], y que tiene aplicaciones prácticas en nuestra vida diaria, desde reducir el tamaño de nuestras imágenes hasta reducir el tamaño de grandes conjuntos de datos. Esto último se ha vuelto relevante en los últimos años, dada la gran cantidad de datos que los instrumentos y observatorios astronómicos producen diariamente [Vohl *et al.*, 2015], y el problema que resulta ser el manejo eficiente de estos datos.

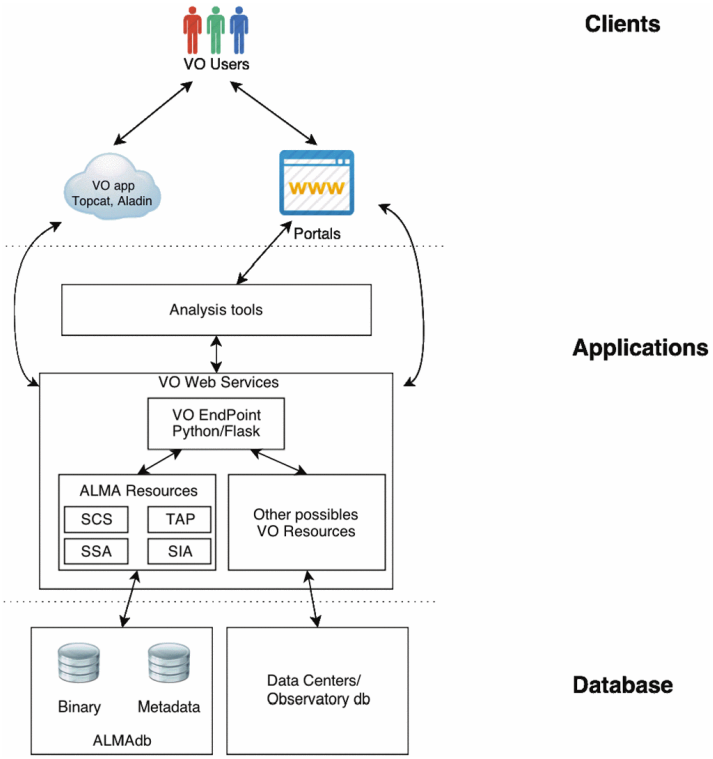


Figura 2: Arquitectura de ChiVO (abstracción de alto nivel).
Fuente: [Solar *et al.*, 2015].

En la actualidad existen múltiples algoritmos de compresión de información [Shanmugasundaram y Lourdusamy, 2011], cada uno con un conjunto de ventajas y desventajas. Típicamente, estas ventajas y desventajas caen en una de dos categorías: tiempo de compresión/descompresión y espacio en memoria. Este *tradeoff* de espacio/-tiempo es algo que es recurrente en la literatura, dado que tiene implicancias prácticas tales como el desempeño de de los buscadores web [Arroyuelo *et al.*, 2012], resaltándose así la importancia de lograr un buen *tradeoff* entre estos dos factores.

El problema que se está abordando en este trabajo no es algo trivial, dado que el método de compresión de documentos a elegir debe ser lo suficientemente eficiente para posibilitar la mantención de la base de datos comprimida en memoria, y lo suficientemente rápido para proporcionar rápido acceso a los documentos que se encuentran en la base de datos. Problemas de este tipo ya se han dado en la literatura, como la compresión de grandes imágenes en astronomía [Bobichon y Bijaoui, 1997] o de grandes volúmenes de datos provenientes de instrumentos astronómicos [Vohl *et al.*, 2015]. Es por esto que la elección de un adecuado algoritmo de compresión es crucial para el óptimo desempeño de las búsquedas a realizar.

2.3. Estructuras de Datos Sucintas

Una estructura de datos sucinta es una estructura que usa una cantidad de espacio que es "cercana" a la cota inferior teórica pero que, a diferencia de otras representaciones comprimidas, sigue permitiendo operaciones de consulta eficientes. El concepto de estructuras sucintas fue inicialmente planteado para codificar vectores de bits, árboles no etiquetados, y grafos planos [Jacobson, 1989]. A diferencia de algoritmos generales de compresión *lossless*, las estructuras sucintas mantienen la característica de poder realizar consultas a las mismas sin la necesidad de ser completamente descomprimidas.

Es justamente por esta última razón que las estructuras sucintas resultan bastante atractivas para dar solución al problema a resolver en el presente trabajo de memoria, dado que permiten mantener datos en memoria principal de forma eficiente, además de permitir consultas a estos datos.

2.3.1. Suffix Arrays

Los arreglos de sufijos, o *Suffix Arrays* [Manber y Myers, 1990] son una estructura que ayuda a resolver una serie de problemas en textos y secuencias, desde compresión y acceso a datos a análisis de secuencias biológicas y descubrimiento de patrones. De acuerdo a [Grossi, 2011], en su forma más simple, los arreglos de sufijos pueden ser vistos como una permutación de los elementos $\{1, 2, ..., n\}$, codificando la secuencia ordenada de sufijos de un texto dado de largo n, en orden lexicográfico. En la Figura 3 se observa un ejemplo de un arreglo de sufijos para la palabra `senslessness$`.

De acuerdo a [Grossi, 2011], construir un arreglo de sufijos para un texto cualquiera puede ser sencillo pero ineficiente, ya que depende fuertemente de la función de ordenamiento utilizada para el ordenamiento lexicográfico. Por ejemplo, si se utiliza la función `qsort` de C en conjunto con una función de comparación que implementada en base a arreglos, la creación del arreglo de sufijos puede tomar tiempo $O(n^2 \log(n))$; por otro lado, se pueden usar algoritmos más sofisticados que toman tiempo $O(n \log(|\Sigma|))$, donde Σ corresponde al alfabeto del texto.

2.3.2. Suffix Trees

Los árboles de sufijos, o *Suffix Trees*, son *tries*[5] comprimidos que contienen todos los sufijos de un texto dado como sus llaves y las posiciones en el texto como sus valores, los cuales posibilitan implementaciones rápidas de muchas operaciones importantes de *strings*. Los árboles de sufijos guardan múltiples relaciones con los arreglos de sufijos, dado que los arreglos de sufijos pueden ser construidos realizando una búsqueda en profundidad sobre el árbol de

[5]Estructura de datos de tipo árbol que permite la recuperación de información.

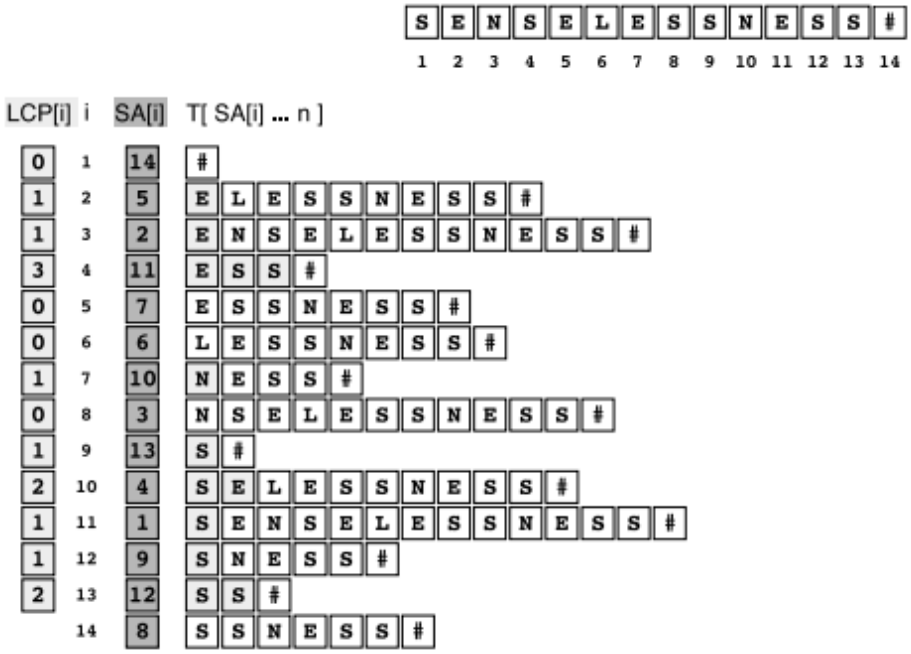


Figura 3: Ejemplo de un *Suffix Array* para la palabra `senslessness$`, en donde el *Suffix Array* corresponde al arreglo `SA[i]`.
Fuente: [Grossi, 2011].

sufijos. El arreglo de sufijos corresponde a los valores de las hojas en el orden en el que fueron visitados durante la búsqueda en profundidad, considerando que los nodos son visitados en el orden lexicográfico de su primer carácter.

Se ha probado que cualquier algoritmo que hace uso de un árbol de sufijos puede ser reemplazado por otro que utiliza arreglos de sufijos en conjunto con información adicional, y que esto resuelve el mismo problema en la misma complejidad de tiempo [Abouelhoda *et al.*, 2004], lo que hace que ambas estructuras sean esencialmente intercambiables. Sin embargo, los arreglos de sufijos presentan mejores complejidades de espacio y resultan generalmente más sencillos de construir que los árboles de sufijos, lo que debe ser tomado en consideración a la hora de elegir uno de estas dos estructuras de datos.

En este contexto, tanto los *Compressed Suffix Arrays* (CSA) como los *Compressed Suffix Trees* (CST) apuntan a reducir la complejidad de espacio de cada estructura a complejidades cercanas a la cota inferior teórica, pero manteniendo un buen desempeño en las operaciones de búsqueda y acceso a la estructura comprimida. En el caso de los *Compressed Suffix Arrays*, y considerando un texto T de n caracteres de un alfabeto Σ, el CSA provee tiempos de búsqueda de $O(m)$ o $O(m+\log(n))$, donde m corresponde al largo de la consulta [Grossi, 2011], además de utilizar espacio $O(nH_k(T)) + o(n)$, donde $H_k(T)$ corresponde a la entropía [Shannon, 1948] empírica de k-esimo orden del texto T.

2.4. La Succint Data Structure Library

La *Succint Data Structure Library*[Gog *et al.*, 2014], o SDSL, es una librería creada por Simon Gog para C++11 que implementa una serie de estructuras sucintas, conteniendo los aspectos más destacados de cerca de 40 publicaciones. Dado que la implementación de estructuras sucintas como las mencionadas anteriormente es en la práctica un proceso no trivial, debido a que frecuentemente requieren operaciones complejas sobre bit-vectores, la librería SDSL provee implementaciones *open-source* y de alta calidad de muchas estructuras sucintas propuestas en la literatura, incluidos los arreglos sufijos comprimidos y árboles sufijos comprimidos que fueron mencionados en el punto anterior.

La SDSL ofrece tres implementaciones de arreglos de sufijos comprimidos, las que se caracterizan por el tipo de estructuras sucintas utilizadas para representar el arreglo de sufijos, su inverso, así como otras estructuras auxiliares (como la *Burrows-Wheeler Table*). Estas implementaciones se resumen en la tabla 2 y tabla 3.

Tabla 2: Implementaciones de *Compressed Suffix Array* disponibles en la SDSL. Fuente: SDSL Cheat Sheet.

Nombre (clase)	Descripción
`csa_bitcompressed`	Basada en SA e ISA almacenadas en *Integer Vectors*.
`csa_sada`	Basada en Ψ (*Successor Array*) almacenada en un *Compressed Integer Vector*.
`csa_wt`	Basada en la *Burrows–Wheeler Transform* (BWT) almacenada en un *Wavelet Tree* (WT).

Tabla 3: Implementaciones de *Compressed Suffix Tree* disponibles en la SDSL. En este contexto, el operador $|\cdot|$ denota el tamaño de la estructura en cuestión. Fuente: SDSL Cheat Sheet.

Nombre (clase)	Descripción
`cst_sada`	Representa un nodo como posición en BPS (*Balanced Parentheses Support*). Operaciones de navegación rápidas, dado que son directamente traducidas en operaciones BPS en la DFS-BPS. Espacio: $4n + o(n) + \|CSA\| + \|LCP\|$ bits.
`cst_sct3`	Representa nodos como intervalos. Rápida construcción, pero operaciones de navegación más lentas. Espacio: $3n + o(n) + \|CSA\| + \|LCP\|$ bits.

Las implementaciones de arreglos de sufijos comprimidos y árboles sufijos comprimidos incluidas en la SDSL, permiten ajustar ciertos parámetros asociados a su construcción y uso, los cuales pueden afectar factores como el tamaño final de la estructura en memoria y el tiempo de acceso a la misma. Estos parámetros modificables por el usuario son resumidos en la tabla 4.

Tabla 4: Parámetros de un arreglo de sufijos comprimido basado en una tabla de Burrows-Wheeler.

Fuente: SDSL Cheat Sheet.

Parámetro	Valor por defecto	Descripción
t_wt	wt_huff<>	Tipo de *Wavelet Tree* a usar por la BWT.
t_dens	32	Densidad de sampleo para valores del *Suffix Array*.
t_dens_inv	64	Densidad de sampleo para valores del *Inverse Suffix Array*.
t_sa_sample_strat	sa_order_sa_sampling<>	Política para el sampleo del *Suffix Array*.
t_isa_sample_strat	isa_sampling<>	Política para el sampleo del *Inverse Suffix Array*.
t_alphabet_strat	*t_wt dependant*	Política para representación del alfabeto.

2.5. Estructuras de soporte

El uso de un CSA o un CST en la SDSL requiere el uso de una estructura de datos de soporte, sobre la cual se almacenarán la tabla BWT, el arreglo de sufijos, entre otros elementos. La elección de la estructura (o estructuras) de soporte es crucial para el óptimo funcionamiento del CSA o CST en la aplicación o problema que se desea abarcar, y tendrá incidencia en el tiempo de ejecución de ciertas operaciones (como *locate* y *extract*) y en el tamaño final de la estructura comprimida en memoria y en disco. En este trabajo, exploraremos 7 estructuras para soportar las operaciones del CSA y el CST, las cuales se pueden agrupar en 2 categorías: aquellas basadas en vectores de bits, y aquellas basadas en *Wavelet Trees*.

2.5.1. Vectores de bits

2.5.1.1. Operaciones básicas

Dado un vector de bits E, se definen las siguientes operaciones básicas sobre el mismo:

1. `E.rank(i)`: Entrega la cantidad de unos hasta la posición `i`, inclusive.
2. `E.select(i)`: Entrega la posición de la `i`-ésima ocurrencia de 1 dentro del vector de bits.
3. `E.access(i)`: Entrega el valor en la `i`-ésima posición del vector de bits.

2.5.1.2. RRR-Vector

Los *RRR-Vectors*, llamados así en honor a sus creadores (Raman, Raman y Rao) [Raman *et al.*, 2007], consisten en una estructura que almacena bits siguiendo un esquema de bloques y superbloques, que apunta a soportar operaciones de `rank` y `select` en tiempo constante $O(1)$, siendo esta última su principal ventaja, sobre todo considerando que las operaciones de rank pueden ser de un largo arbitrario.

2.5.1.3. SD-Vector

Los SD-Vectors, propuestos por [Okanohara y Sadakane, 2007], son una estructura que comprime vectores de bits poco densos (esto es, un vector de bits en donde la cantidad de 1s presente es considerablemente menor a la cantidad de ceros) al representar las posiciones de unos mediante la representación de Elías-Fano para secuencias no decrecientes. Esta estructura soporta operaciones de rank y select en tiempo constante, mientras que las operaciones de access son soportadas en tiempo logarítmico $O(\log \frac{n}{m})$, dependiente del largo n del vector de bits y de la cantidad m de unos presentes en el mismo.

2.5.2. Wavelet Trees

Los *Wavelet Trees* [Grossi *et al.*, 2003] son, en su forma más esencial, árboles compuestos por vectores de bits, y son generalmente utilizados para comprimir representaciones BWT de cadenas de caracteres [Ferragina *et al.*, 2009]. Un *wavelet tree* particiona recursivamente una cadena de caracteres s en 2 partes (particionando su alfabeto) hasta que nos encontremos con particiones que contengan sólo 1 o 2 símbolos (que pasarán a ser las hojas del árbol), transformando así una entrada s en una jerarquía de vectores de bits. Esto permite que los *wavelet trees* puedan ser usados para procesar queries de rango de forma eficiente. En promedio, una operación de rank es soportada con tiempo $O(\log A)$, donde A es el largo del alfabeto utilizado.

2.5.2.1. Operaciones básicas

Dado un wavelet tree T, se definen las siguientes operaciones básicas sobre el mismo:

1. `T.rank(c,i)`: Entrega la cantidad de ocurrencias del caracter `c` hasta la posición `i` dentro del árbol, inclusive.
2. `T.select(c,i)`: Entrega la posición de la `i`-ésima ocurrencia del caracter `c` dentro del árbol.
3. `T.access(i)`: Entrega el caracter en la `i`-ésima posición del árbol.

2.5.2.2. Run-Length Wavelet Tree

En [Mäkinen y Navarro, 2005], los autores proponen una modificación a los Wavelet Trees,

planteando la inclusión del Run-Length Encoding al proceso de creación de dicho árbol. En lugar de actuar directamente sobre la cadena de caracteres, los autores plantean actuar sobre los runs de caracteres que ocurren sobre la cadena de caracteres. Ello produce 2 arreglos, un arreglo S que contiene un carácter por cada run de la cadena de caracteres, y un arreglo B de n bits que delimita el comienzo de cada run de caracteres. Esto significa que, para textos que cuentan con una gran cantidad de runs, se pueden lograr mejores compresiones que el Wavelet Tree original.

2.5.2.3. Alphabet Partitioning

En [Barbay *et al.*, 2014], los autores proponen un estructura que almacena una cadena de caracteres s (con un alfabeto de tamaño σ) utilizando espacio de orden lineal, soportando operaciones rank, select y access en un tiempo de peor caso de $O(\log \log \sigma)$, siendo este tiempo logarítmico en el caso promedio. Esta estructura es práctica en alfabetos grandes, y se basa en la partición de dicho alfabeto en caracteres de frecuencia similar. Las subsecuencias formadas por caracteres de frecuencia similar pueden ser codificadas utilizando representaciones de rápido acceso, como GMR.

2.5.2.4. Wavelet Matrix

Esta variante de los Wavelet Trees, propuesta por [Claude *et al.*, 2015], plantea que en casos en los que el largo del alfabeto es considerable comparado al largo del texto, las representaciones de Wavelet Trees actuales incurren en notables overheads de espacio o tiempo. Frente a esto, los autores proponen una representación alternativa de los Wavelet Trees para grandes alfabetos, la cual reordena los nodos en cada nivel del árbol (es decir, no almacena los dos hijos de un nodo v de forma alineada con v). De esta forma, esta estructura retiene las propiedades de los Wavelet Trees y logra que los recorridos necesarios para soportar las operaciones sean simplificados y acelerados, lo que la hace considerablemente más rápida bajo las circunstancias mencionadas anteriormente.

2.5.2.5. Huffman-shaped Wavelet Tree

Los mismos autores [Mäkinen y Navarro, 2005] que propusieron la modificación de los Wavelet Trees mediante la inclusión del Run-Length Encoding al proceso de creación del árbol, también proponen como una segunda alternativa cambiar la forma del Wavelet Tree de forma que se ajuste a una codificación de Huffman. De esta forma, en lugar de utilizar un árbol binario balanceado, se utiliza el Huffman Tree de la cadena de caracteres s para definir la forma del Wavelet Tree correspondiente. Esto implica que la posición de un carácter de la cadena s estará directamente relacionada a la frecuencia con la que dicho carácter ocurre dentro de la misma, por lo que caracteres más frecuentes estarán ubicados en hojas más cercanas a la raíz del Wavelet Tree.

2.5.3. GMR

En [Golynski *et al.*, 2006], los autores plantean una estructura (llamada así por los apellidos de sus autores) que permite soportar operaciones rank, select y access en cadenas de caracteres con alfabetos σ grandes, al igual que la Wavelet Matrix. Las operaciones rank y access son soportadas por esta estructura con tiempo $O(\log \log \sigma)$, mientras que las operaciones select son soportadas con tiempo $O(1)$.

Para lograrlo, plantean el uso de una matriz T de dimensiones $t \cdot n$, en donde una posición $T[c, i]$ indica si c ocurre o no en la posición i de la cadena de caracteres s, de largo n. Una concatenación de las filas de esta matriz nos entrega un vector de bits A, sobre el cual se pueden soportar operaciones de rank y select respecto a la cadena de caracteres original. De la concatenación A se obtienen bloques de tamaño t, sobre los cuales es posible soportar operaciones rank y select de forma restringida, además de poder determinar la cardinalidad de uno de estos bloques. Con esto, es posible definir un vector de bits unario B con las cardinalidades de todos los bloques de A. Finalmente, el soporte de operaciones rank y select se realiza con una combinación de operaciones rank y select sobre las estructuras descritas anteriormente.

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

En esta sección, ahondaremos en la solución propuesta para la compresión y consulta en memoria de recursos del Observatorio Virtual. Subdividiremos esta sección en las siguientes subsecciones:

- Determinación de forma de procesamiento de *queries*.
- Determinación de estructura sucinta a utilizar.
- Diseño de la arquitectura del indexador.

3.1. Consultas a procesar

Los tipos de consultas que deberán ser soportadas por el prototipo a realizar deberán responder a las necesidades de los usuarios del Observatorio Virtual Chileno. En el Capítulo 1, se determinó que la problemática a resolver es la búsqueda de servicios relevantes dentro del Observatorio Virtual a partir de una query especificada por el usuario, por lo que es necesario que el prototipo a desarrollar soporte este tipo de consultas.

Una de las formas para determinar si un recurso en particular es relevante para la query indicada por el usuario es la presencia de uno o más elementos de la query dentro del recurso. La frecuencia con la que estos elementos se encuentran en un recurso también es un fuerte indicador de la relevancia del mismo para la query utilzada. Sin embargo, la utilización de estos dos indicadores de forma exclusiva no garantiza que un recurso específico sea o no relevante para lo que se está buscando, sino que también es necesario determinar el contexto en el que los elementos de la query aparecen dentro del recurso.

Para ilustrar esto último, consideremos la palabra "`arroyuelo`". Dependiendo del contexto en el que se esté utilizando, "`arroyuelo`" puede ser interpretado como un apellido (Diego Arroyuelo), una localidad (Arroyuelo, España) o como un cuerpo de agua. En el contexto del Observatorio Virtual, una búsqueda por "`solar`" puede ser entendida como la búsqueda de un apellido (Mauricio Solar), o como la búsqueda de un cuerpo astronómico (Sistema Solar).

Por lo tanto, además de considerar la presencia o no de la query utilizada dentro de un recurso, y la frecuencia con la que esta query aparece dentro del mismo, es necesario determinar en qué contexto (o contextos) está presente la query dentro del recurso. En el caso del Observatorio Virtual, el contexto de un dato dentro del un recurso está dado por la columna en la que este dato está presente. Tomando el ejemplo de "`solar`" presentado anteriormente, esto significa que, dentro de un recurso, la palabra "`solar`" puede aparecer bajo la columna "`autor`" o bajo la columna "`obj_observado`".

A raíz de lo anterior, el prototipo a desarrollar deberá entregar, para una query y recurso dados, las columnas en las cuales la query se encuentra presente dentro del recurso, y la cantidad de veces que dicha query ocurre dentro del mismo.

3.2. Determinación de las fuentes de datos

Como se habló en el Capítulo 2, una de las fuentes de datos disponibles en el Observatorio Virtual es el *Table Access Protocol* (TAP), parte de la *Data Access Layer*. Este protocolo nos permite obtener una tabla de datos, en formato VOTable (un formato basado en XML), en base a una consulta previamente realizada al servicio en cuestión.

Actualmente, existen cerca de 150 servicios TAP publicados en el *Registry of Registries* de la IVOA, lo que representa una cantidad muy alta para abarcar durante este trabajo. Es por esto que se decidió acotar el alcance de los datos a procesar durante este trabajo a un subgrupo de estos servicios, a forma de agilizar los experimentos realizados y reducir la cantidad de datos a un volumen manejable.

3.3. Estructura sucinta a utilizar

Debido a que esta aplicación en particular implica la realización de búsquedas de substrings dentro de colecciones de metadatos, se procede a utilizar un CSA, y a estudiar la combinación óptima de estructuras de soporte para este problema.

Para realizar las pruebas, y como una forma de estandarizar las mediciones a través del proceso de desarrollo de la solución, se decidió crear una serie de *queries* con las palabras más frecuentes del archivo.

Para poder categorizar las distintas *queries* obtenidas, se requiere de un factor que indique la dificultad de la misma para ser procesada por nuestro sistema. En este contexto, se entiende por dificultad como el tiempo y cantidad de operaciones de extracción necesarias para obtener los datos necesarios que permitan determinar las columnas en las que se encuentra presente nuestra *query*, y la cantidad de veces que se encuentra presente en cada una de ellas. Para estos efectos, se definió un *Access Ratio* de la siguiente forma:

$$\text{Access Ratio} = \frac{\text{Cantidad de operaciones de extract}}{\text{Número de columnas que contienen la query}}$$

Este factor nos permite clasificar nuestras *queries* en 3 categorías de dificultad: Fácil, Medio y Difícil, las que están directamente relacionadas con el tiempo que será necesario para procesar la consulta y entregar un resultado.

3.4. Estructura del documento a comprimir

En primer lugar, es necesario determinar la forma del documento que se almacenará en memoria, sabiendo que la estructura del documento que se consultará determinará directamente la forma (y consecuentemente, los tiempos) en las que se procesarán las consultas y se computarán las respuestas.

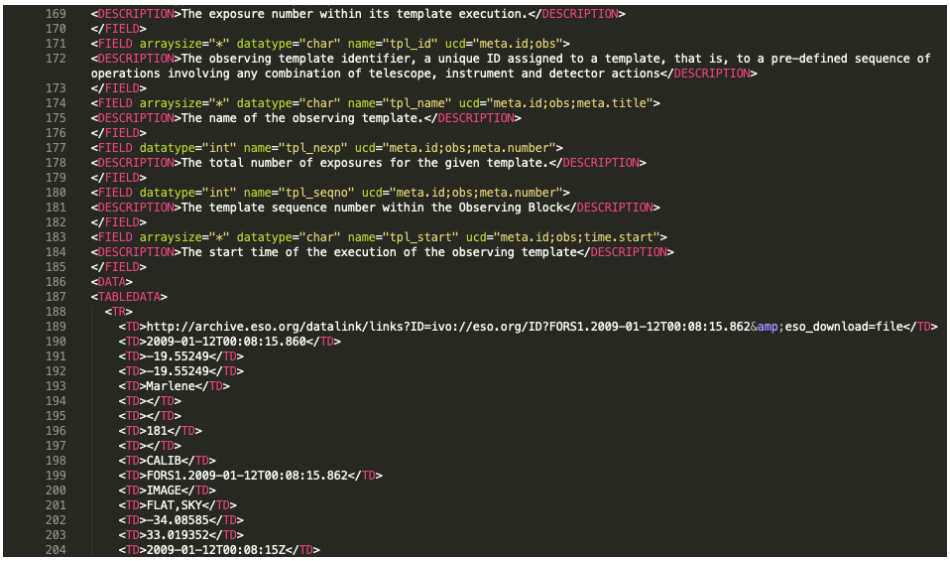
Para el almacenamiento de los documentos en memoria, esencialmente se cuenta con 2 opciones: almacenar el documento "*as-is*", conservando su estructura y forma original, u optimizar el documento para el uso con alguna estructura sucinta. Estas opciones tienen una serie de ventajas y desventajas, las que se analizan a continuación. Almacenar el documento sin modificaciones en memoria tiene la ventaja de conservar toda la información (incluidos los metadatos) contenida en el mismo, lo que posibilita la realización de análisis no contemplados inicialmente, dado que se cuenta con toda la información original almacenada en la estructura sucinta en memoria. La principal desventaja de este enfoque es que no toma en cuenta las particularidades o fortalezas de la estructura sucinta que se utiliza, lo que limita la solución a una más genérica.

Por otro lado, la modificación del documento original permitiría el aprovechamiento de las fortalezas de las estructuras sucintas que se desean utilizar, adaptando el documento para optimizar las consultas que se desean soportar sobre el mismo. Esto trae la desventaja que no se podrían soportar otras consultas más que las establecidas inicialmente, o que requieran más datos (o metadatos) que los incluidos en el documento modificado almacenado en memoria.

Dado que en esta oportunidad se conoce de antemano el tipo de consultas que se procesarán, y dado que nuestro objetivo es maximizar la eficiencia de consultas en memoria, se procede con la segunda opción.

En la sección anterior, se discutieron las fuentes de los datos que se utilizaron en este trabajo para la realización de experimentos y obtención de resultados. Un extracto del contenido de uno de estos archivos se muestra en la figura 4. En este extracto se puede observar que mucha de la información presente en el archivo corresponde a delimitadores y metadatos propios de un archivo basado en la estructura XML, y que no son relevantes para el problema en estudio, por lo que pueden ser removidos. De esta forma, es posible obtener un archivo sólo con la información esencial para procesar las consultas requeridas, contribuyendo así a una mayor eficiencia en tiempo y en espacio.

Dado que las consultas que deberán ser respondidas por nuestra solución se basan en indicar las columnas en las cuales se encuentra presente nuestra consulta, además de la cantidad de veces que esto ocurre por cada columna, es importante contar con el nombre de la columna para cada dato presente en el archivo; en otras palabras, es necesario incluir junto con el dato el nombre de la columna a la que pertenece. De esta forma, es posible prescindir de la estructura XML del archivo junto con el resto de sus metadatos, reduciendo el *overhead*



```
<DESCRIPTION>The exposure number within its template execution.</DESCRIPTION>
</FIELD>
<FIELD arraysize="*" datatype="char" name="tpl_id" ucd="meta.id;obs">
<DESCRIPTION>The observing template identifier, a unique ID assigned to a template, that is, to a pre-defined sequence of operations involving any combination of telescope, instrument and detector actions</DESCRIPTION>
</FIELD>
<FIELD arraysize="*" datatype="char" name="tpl_name" ucd="meta.id;obs;meta.title">
<DESCRIPTION>The name of the observing template.</DESCRIPTION>
</FIELD>
<FIELD datatype="int" name="tpl_nexp" ucd="meta.id;obs;meta.number">
<DESCRIPTION>The total number of exposures for the given template.</DESCRIPTION>
</FIELD>
<FIELD datatype="int" name="tpl_seqno" ucd="meta.id;obs;meta.number">
<DESCRIPTION>The template sequence number within the Observing Block</DESCRIPTION>
</FIELD>
<FIELD arraysize="*" datatype="char" name="tpl_start" ucd="meta.id;obs;time.start">
<DESCRIPTION>The start time of the execution of the observing template</DESCRIPTION>
</FIELD>
<DATA>
<TABLEDATA>
  <TR>
    <TD>http://archive.eso.org/datalink/links?ID=ivo://eso.org/ID?FORS1.2009-01-12T00:08:15.862&amp;eso_download=file</TD>
    <TD>2009-01-12T00:08:15.860</TD>
    <TD>-19.55249</TD>
    <TD>-19.55249</TD>
    <TD>Marlene</TD>
    <TD></TD>
    <TD></TD>
    <TD>181</TD>
    <TD></TD>
    <TD>CALIB</TD>
    <TD>FORS1.2009-01-12T00:08:15.862</TD>
    <TD>IMAGE</TD>
    <TD>FLAT,SKY</TD>
    <TD>-34.08585</TD>
    <TD>33.019352</TD>
    <TD>2009-01-12T00:08:15Z</TD>
```


Figura 4: Extracto de un archivo VOTable obtenido mediante consulta a un servicio TAP perteneciente al Observatorio Virtual de ESO.
Fuente: Elaboración propia.

de espacio y produciendo un archivo que contiene únicamente información relevante para el procesamiento de las consultas necesarias.

Se puede ir un paso más allá y aprovechar las particularidades del CSA que se utilizará para guardar y consultar esta información en memoria. Dado que las búsquedas que se realizarán sobre el CSA serán (en esencia) búsquedas de prefijos en el texto, se puede modificar dicho archivo para facilitar la creación del arreglo de sufijos, ordenando previamente el archivo de forma lexicográficamente creciente sobre los datos contenidos en el archivo. Además, esta modificación del archivo mediante el ordenado lexicográfico de sus elementos expondrá de forma directa los datos repetidos dentro de nuestro archivo, quedando todos ellos contiguos dentro del mismo. La presencia de estas cadenas de datos repetidos ocasionarán que el archivo sea más compresible, dado que el CSA es capaz de aprovechar estas cadenas de repeticiones mediante el arreglo de sufijos. De esta forma, la estructura del archivo intermedio queda como se muestra en la figura 5.

3.5. Método de proceso de consultas

Junto a la determinación de la estructura del documento que se mantendrá en memoria, y sobre el cual se realizarán las búsquedas para procesar una consulta dada, es necesario determinar el algoritmo que se deberá seguir para procesar y dar respuesta a dicha consulta. Al respecto, se han planteado 2 algoritmos para realizar el procesado de las consultas, uno de ellos de carácter más genérico y uno adaptado para el CSA que se utiliza para almacenar el documento en memoria.

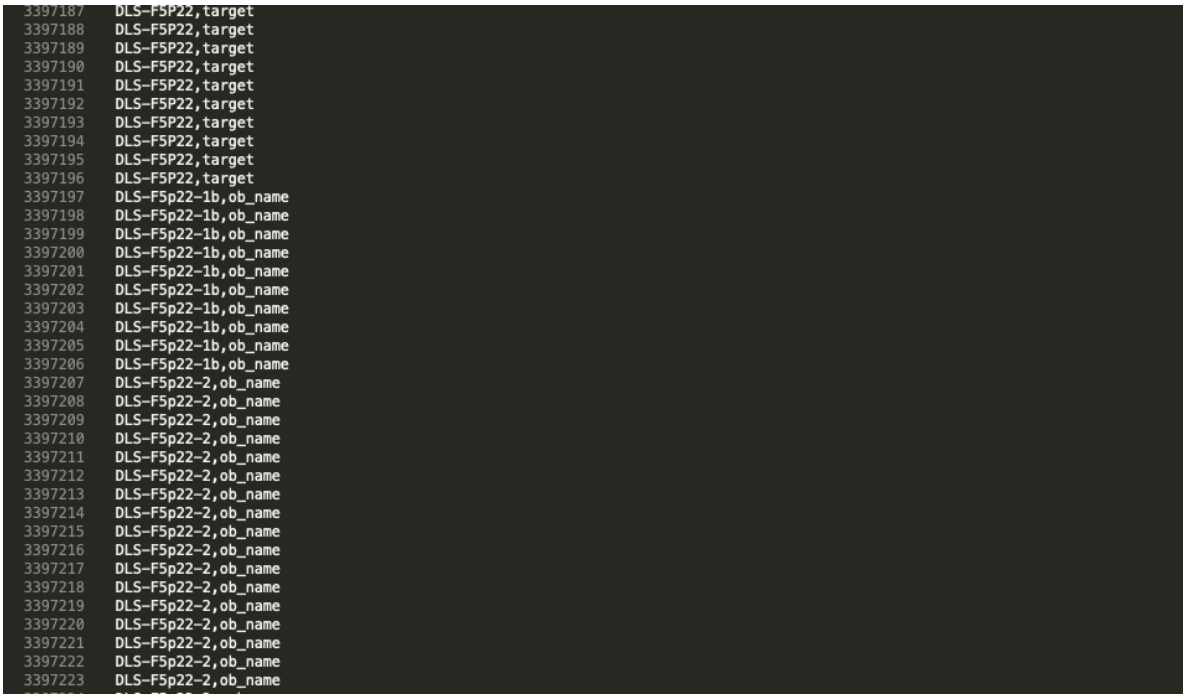


Figura 5: Extracto del archivo intermedio con las modificaciones propuestas. Aquí es evidente la presencia de cadenas de repeticiones, lo que favorece la compresión del mismo por parte del CSA.

Fuente: Elaboración propia.

El primer método se describe en Algoritmo 1. Este método, al ser el primer método estudiado, utiliza un archivo intermedio con una arquitectura distinta a la presentada anteriormente, en donde el nombre de la columna se incluye al comienzo de la fila, dentro de un tag <TR>. En esencia, este método cuenta y obtiene todas las ocurrencias de la consulta dentro del documento y, para cada ocurrencia, obtiene la línea completa que la contiene (buscando por los delimitadores <TR> y </TR> al inicio y fin de la línea, respectivamente). De esta forma se obtiene el dato particular y la columna a la que este dato pertenece, lo que posibilita realizar la agregación de ocurrencias por columna y entregar el resultado al usuario.

Por otro lado, el enfoque optimizado al CSA que se utiliza para mantener el documento en memoria utiliza funciones específicas de esta estructura, así como un nuevo enfoque para determinar la cantidad de ocurrencias por columna, con el objetivo de mejorar los tiempos de respuesta de consultas. Este método se presenta en Algoritmo 2. El mismo utiliza la estructura de archivo comprimido descrita en la sección anterior, y se diferencia del método genérico en que, para encontrar el número de ocurrencias de la consulta (tanto a nivel general como por cada columna), utilizamos directamente el arreglo de sufijos del CSA.

Sea SA el arreglo de sufijos correspondiente al documento comprimido utilizando un CSA, y Q una consulta realizada por el usuario sobre el documento (en este caso, representado por el archivo intermedio discutido en el punto anterior). En primer lugar es necesario de-

Data: Consulta del usuario (Q), documento comprimido (fm_index).
Result: Columnas que contienen ocurrencias de la consulta, junto con el número de ocurrencias por columna.

```
1  cuenta = Dict;
2  while existe Q do
3      cuenta = count(Q, fm_index);
4      ubicaciones = locate(Q, fm_index);
5      for ubicaciones do
6          Obtener inicio y término de línea completa que contiene a Q;
7          linea = extract(fm_index, inicio, fin);
8          Obtener nombre de columna desde linea;
9          Actualizar cuenta de columna;
10     end
11 end
```

Algorithm 1: Método genérico para procesamiento de consultas utilizando funciones de alto nivel de la SDSL.

terminar las ubicaciones de Q dentro del documento, lo que se logra obteniendo el rango en SA para el cual Q es un prefijo, mediante el uso de la función `lex_extract` de la SDSL. Esto nos permite determinar un subconjunto de SA, S_{ij}, que corresponde a las ubicaciones de Q dentro del archivo intermedio.

Posteriormente, usando el resultado anterior se procede a obtener los nombres de las columnas que se encuentran dentro del rango S_{ij}, que corresponden a todas las columnas que contienen ocurrencias de Q, y la cantidad de ocurrencias en cada una. Esto se realiza determinando los subrangos dentro de S_{ij} para los cuales el texto original correspondiente corresponde a un prefijo:

1. Se obtiene el texto original a partir de la primera ocurrencia del rango (que, en el caso inicial, corresponde al texto original entre las posiciones $CSA[S_i]$ y el final de la línea correspondiente, la cual contiene a la consulta Q, utilizando la función `extract` de la SDSL. A partir de esta línea extraída, se puede obtener el nombre C de la columna que contiene al dato (y, por ende, consulta Q) presente en esta línea en el documento original.

2. Con la sección de la línea extraída en el paso anterior se determina el subrango S_{kl}, subconjunto de S_{ij} que corresponde a las ocurrencias de dicha línea en S_{ij} (o, dicho de otra forma, el subrango de s_{ij} para el cual la línea extraída del paso anterior corresponde a un prefijo). El tamaño de este subrango indicará la cantidad de ocurrencias de la consulta Q dentro de la columna C, las que son actualizadas en la cuenta general para la consulta Q.

3. Posteriormente, se continúa con el siguiente subrango, partiendo con la posición inicial $l + 1$ y repitiendo el ciclo mientras $l <= j$.

De esta forma, se reducen al mínimo las operaciones de extract sobre el CSA, que corresponden a operaciones considerablemente más costosas que operaciones de *locate* y *extract* sobre el mismo, además de aprovechas las características del arreglo de sufijos para realizar el procesado de la consulta de forma más rápida y eficiente.

Data: Consulta del usuario (Q), documento comprimido (fm_index).
Result: Columnas que contienen ocurrencias de la consulta, junto con el número de ocurrencias por columna.

```
1  cuenta = Dict;
2  while existe Q do
       /* Obtención del rango de consulta en SA                          */
3      rango = lex_extract(fm_index, Q);
4      pos_actual = rango[inicial];
5      while pos_actual < rango[final] do
6          Obtener fin de línea completa que contiene a Q;
7          linea = extract(fm_index, pos_actual, fin);
8          Obtener nombre de columna desde línea;
           /* Obtener rango de columna conteniendo consulta en SA        */
9          rango_col = lex_extract(fm_index, linea);
10         Actualizar cuenta de columna en rango_col[fin]-rango_col[inicio];
11     end
12 end
```

Algorithm 2: Método optimizado para procesamiento de consultas utilizando funciones de alto nivel de la SDSL y específicas del CSA.

3.6. Arquitectura del buscador

Para la implementación del prototipo, se propone el desarrollo de 4 subsistemas:

1. Un **rastreador** o *crawler*, que se encargue de realizar las consultas necesarias a los servicios TAP especificados en la sección 3.2, obteniendo así todos los datos (en formato VOTable) ofrecidos por dicho servicio.

2. Un **indexador**, que se encargue de tomar los datos en formato VOTable obtenidos por el rastreador y realice las operaciones descritas en las secciones anteriores para producir un índice comprimido, que posteriormente podrá ser consultado.

3. Uno o más **buscadores**, que se encargarán de procesar una consulta dada sobre uno o más índices comprimidos producidos por el indexador.

4. Un **despachador** o *dispatcher*, que recibirá la consulta del usuario y se encargará de consultar a los diferentes **buscadores** de forma distribuida, armando así la respuesta que se entregará al usuario.

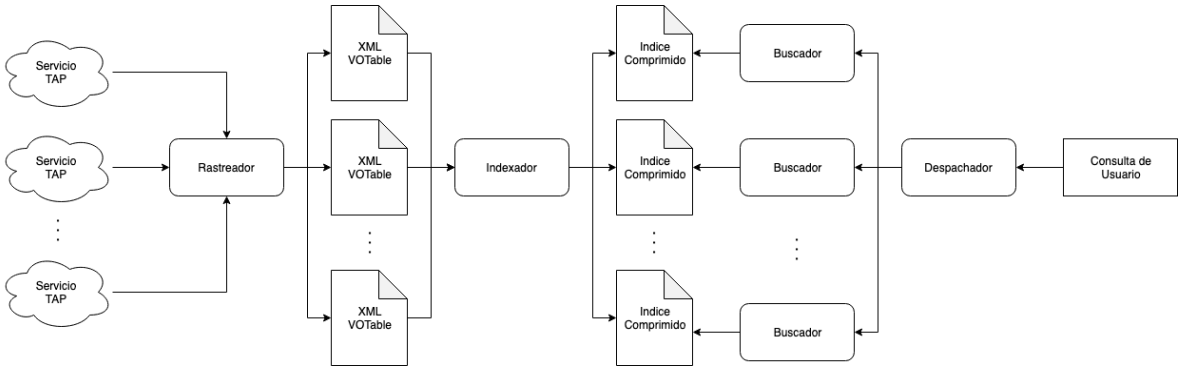


Figura 6: Diagrama de la arquitectura propuesta para el prototipo de buscador.
Fuente: Elaboración propia.

Un resumen de esta arquitectura se presenta en la figura 6. El indexador y el buscador se desarrollan en C++, debido a que la librería SDSL utilizada para la compresión y manejo de las estructuras comprimidas funciona en C++, y porque estos dos módulos son críticos a la hora de crear los índices y acceder a ellos de forma eficiente, por lo que lograr el máximo desempeño es crucial. Por otro lado, el rastreador y el despachador se desarrollan en Python, principalmente por su facilidad para el manejo de archivos y conexión con servicios web.

Dado que el objetivo central de este trabajo es lograr una compresión de documentos para búsquedas eficientes en memoria de los recursos del Observatorio Virtual, el foco estará puesto únicamente en el indexador y buscador para realizar las validaciones necesarias, entendiendo que es en estos dos módulos donde se encuentra el grueso de la implementación del prototipo de buscador.

CAPÍTULO 4
VALIDACIÓN DE LA SOLUCIÓN

En esta sección, se presentarán los resultados que validan la solución propuesta. En primera instancia se mencionarán las consideraciones previas a la realización de los experimentos, tales como las características del hardware utilizado para su realización, para luego mencionar las diferentes pruebas realizadas para validar la solución propuesta, finalizando con los resultados obtenidos de las pruebas anteriores.

4.1. Consideraciones previas

Excepto cuando se mencione lo contrario, todas las pruebas fueron realizadas en un servidor HP ProLiant DL380p Gen8, con un CPU Intel(R) Xeon(R) E5-2630, 12 Cores @ 2.30GHz, 15MB L3 Cache y 94 GB RAM.

Adicionalmente, cuando se haga referencia a espacio o a factor de espacio durante los experimentos, se estará haciendo referencia al siguiente factor:

$$\text{espacio} = \frac{\text{Tamaño de archivo comprimido}}{\text{Tamaño de archivo original}}$$

4.2. Set de datos utilizados

Para realizar los experimentos planteados más adelante, se utilizaron los siguientes datasets:

1. **Archivo TAP**: Archivo XML de 141MB, en formato VOTable, el cual contiene 57 columnas y 100000 filas que corresponden a observaciones realizadas en el observatorio de Paranal de la ESO, obtenido directamente desde el servicio TAP `TAP_OBS` de la misma organización.
2. **Dataset TAP**: Dataset de más de 40GB de datos, distribuidos en 259 archivos XML, en formato VOTable, los cuales contienen un máximo de 500000 filas correspondientes a datos obtenidos de 5 servicios TAP previamente seleccionados:
    - GaVO DC TAP, el servicio TAP del data center de GaVO, el observatorio virtual alemán. Algunos de los archivos XML más grandes provienen de este servicio.
    - ChiVO TAP, el servicio TAP del Observatorio Virtual Chileno. Aunque fue inicialmente considerado como una de las fuentes, el servicio presentó problemas durante la obtención de este dataset, por lo que tuvo que ser omitido.

- ESO TAP_OBS, un servicio para acceder a datos brutos y reducidos, así como mediciones ambientales, del La Silla Paranal Observatory.
- GAIA, servicio TAP del proyecto GAIA operado por la ESA.
- Hubble/TAP, servicio TAP del archivo europeo del telescopio espacial Hubble.
- ALMA (NAOJ), el servicio de Observatorio Virtual de ALMA operado por NAOJ.

4.3. Experimentos realizados

Obtención de corpus de consultas: En primer lugar, es necesario determinar un corpus de consultas sobre las cuales realizar nuestros experimentos. Dado que, a nuestro entendimiento, no existe un cuerpo de consultas realizadas al Observatorio Virtual que sea obtenible y utilizable, se hace necesaria la obtención de un cuerpo de consultas para realizar nuestros experimentos. Esto se realiza obteniendo las palabras más frecuentes contenidas dentro del archivo a utilizar, y clasificando cada palabra de acuerdo a su *access ratio*, el cual se definió en la Sección 3 como:

$$\text{Access Ratio} = \frac{\text{Cantidad de operaciones de extract}}{\text{Número de columnas que contienen la query}}$$

Así, se definen 3 categorías de consultas dependiendo del *access ratio* de las consultas: Alta complejidad, Mediana complejidad, y Baja complejidad, cuyos valores quedarán definidos en base a los *access ratios* de las consultas obtenidas mediante el procedimiento descrito anteriormente.

Evaluación de métodos de procesado de consultas: En segunda instancia, es necesario establecer las diferencias en desempeño de tiempo de las 2 alternativas descritas en la Sección 3.5, esto es, utilizar un método más simple (pero más genérico) v/s utilizar un método ligeramente más complejo, pero que aprovecha las características y fortalezas de la estructura sucinta utilizada. Para esto, se utilizará el Archivo TAP sobre un conjunto de 3 consultas previamente determinadas, extraídas del corpus antes mencionado.

Determinación de método óptimo de compresión: Luego de lo anterior, es necesario determinar el mejor método de compresión de los datos utilizados, para lo que es necesario probar múltiples combinaciones de estructuras de soporte, parámetros y consultas para conocer las estructuras más y menos idóneas para esta tarea. Estas estructuras, junto a su equivalente en la SDSL, se listan en la Tabla 5. Para determinar la mejor estructura para esta aplicación se utilizará el corpus de consultas definido anteriormente, variando la densidad de muestreo de cada estructura en potencias de 2, comenzando con una densidad de muestreo de 8 y terminando con una densidad de muestreo de 2048.

Determinación de baseline: Además, es necesaria la determinación de un baseline, contra el cual se compararán los resultados obtenidos. Dado que la suite GaVO DACHS[6], el entorno

[6] http://docs.g-vo.org/DaCHS/

Tabla 5: Estructuras evaluadas para la solución propuesta, junto a su equivalente en la SDSL. Todas las estructuras aquí listadas están basadas en Wavelet Trees.
Fuente: Elaboración propia.

Nombre (clase SDSL)	Descripción
`wt_rlmn<>`	Run-Length Encoding
`wt_rlmn<rrr_vector>`	Run-Length Encoding, parametrizada con RRR-Vector
`wt_ap<>`	Alphabet Partitioning
`wt_ap<wm_int, wm_int>`	Alphabet Partitioning, parametrizada con Wavelet Matrix
`wt_ap<wm_int, wt_gmr>`	Alphabet Partitioning, parametrizada con Wavelet Matrix y GMR
`wt_ap<wt_huff, wt_huff>`	Alphabet Partitioning, parametrizada con Huffman-Shaped Wavelet Tree
`wm_int<>`	Wavelet Matrix
`wm_int<rrr_vector>`	Wavelet Matrix, parametrizada con RRR-Vector
`wm_int<sd_vector>`	Wavelet Matrix, parametrizada con SD-Vector
`wt_huff<>`	Huffman-Shaped Wavelet Tree

de software utilizado por ChiVO, utiliza un backend PostgreSQL para mantener los metadatos de los datos contenidos dentro de sus servicios, se decidió obtener un baseline basado en este motor de base de datos. Para ello se construyó una query SQL que, utilizando la misma fuente de datos de los experimentos descritos en el párrafo anterior, obtuviese los mismos resultados que los obtenidos por la solución propuesta. La query utilizada es la indicada en el Listing 1.

Listing 1: Query SQL utilizada para obtener el baseline de espacio y tiempo para esta aplicación.

```
SELECT column_name, COUNT(column_name) AS CountOf
FROM public."{TABLE}"
WHERE value LIKE '%{QUERY}%'
GROUP BY column_name;
```

Para obtener los baselines de espacio y tiempo, esta query fue procesada en una instalación de PostgreSQL 9.4 sobre una máquina virtual equipada con un Intel(R) Xeon(R) CPU E5-2630 v3 y 32GB de RAM. Considerando que esta máquina virtual posee un CPU de la misma familia (y modelo) de procesador que el encontrado en el servidor HP mencionado en la Sección 4.1, y dado que PostgreSQL no utilizará más de 32GB de RAM para mantener la tabla en memoria y procesar consultas, es posible considerar estas dos máquinas como equivalentes, pudiendo comparar los resultados obtenidos por la utilización de cada una de estas.

Aplicación práctica - Prototipo de buscador: Finalmente, se analizará la aplicación práctica planteada para el método de compresión discutido en este trabajo, que consiste en la elaboración de un prototipo de motor de búsqueda sobre una colección de datos obtenidos de

un conjunto seleccionado de servicios TAP del Observatorio Virtual.

4.4. Resultados

4.4.1. Obtención de corpus de consultas

Se obtuvieron todas las palabras contenidas en el Archivo TAP, y se obtuvieron sus frecuencias. Luego, se seleccionaron las primeras 200 palabras más frecuentes, algunas de las cuales se muestran en la Tabla 6.

Tabla 6: Frecuencias de las 10 palabras más frecuentes del Archivo TAP.
Fuente: Elaboración propia.

Palabra	Frecuencia
”	1093573
’NaN’	793223
’FORS2’	371196
’0’	257352
’1’	250759
’2003’	203601
’2002’	192307
’11’	154432
’FORS’	143286
’60’	140689

Llama la atención el hecho que la “palabra” más frecuente del Archivo TAP sea un espacio vacío, así como la presencia de números dentro de las 10 palabras más frecuentes dentro del archivo (esto, dado que se está considerando toda cadena de caracteres separada por “-”, “,”, “_” o “.” como una palabra). Debido a esto, se decidió descartar las palabras que tienen la menor probabilidad de formar parte de una consulta. Bajo esta lógica, es más probable que las palabras “ENGINEERING” o “‘solar” sean utilizadas en una consulta en lugar de “http://archive” o “0”, que es una de las palabras obtenidas mediante este método. Un extracto de las 69 palabras obtenidas luego de este filtro se muestra en la Tabla 7.

Una vez que se tienen estas palabras, es necesario calcular su Access Ratio para categorizarlas de acuerdo a su complejidad, utilizando la relación presentada en la sección anterior:

$$\text{Access Ratio} = \frac{\text{Cantidad de operaciones de extract}}{\text{Número de columnas que contienen la query}}$$

Por lo tanto, para cada palabra obtenida es necesario obtener la cantidad de operaciones de extract requeridas para procesar la consulta y la cantidad de columnas en el documento que

Tabla 7: Frecuencias de las 10 palabras más frecuentes del Archivo TAP, filtrando palabras improbables de ser utilizadas en una consulta.
Fuente: Elaboración propia.

Palabra	Frecuencia
'NaN'	793223
'FORS2'	371196
'FORS'	143286
'TEAM'	134982
'eso'	100000
'fits'	100000
'IMG'	91632
'ESO'	88327
'VLT'	86042
'CALIB'	72936

contienen a la consulta, utilizando para ello el método descrito en la Sección 3. De esta forma, se logra obtener el Access Ratio para cada palabra dentro de nuestro corpus, lo que permite la clasificación de las consultas en las 3 categorías propuestas de acuerdo a su complejidad. Un extracto de esto se presenta en la Tabla 8.

Tabla 8: Frecuencias y Access Ratios de las 3 palabras con mayor Access Ratio por cada categoría de consulta.
Fuente: Elaboración propia.

Palabra	Frecuencia	Access Ratio	Complejidad
'eso'	100000	100009.00	Alta
'FORS2'	371196	33809.50	Alta
'FORS'	143286	29189.70	Alta
'GALAXIES'	10799	53.00	Media
'CLUSTER'	4296	39.00	Media
'SPECTROSCOPY'	2728	28.00	Media
'HISTORY'	1684	9.00	Baja
'GROUP'	1236	8.00	Baja
'Imaging'	14432	7.50	Baja

Para separar las palabras en categorías de acuerdo a su complejidad, se definieron criterios en base a la tendencia observada de los resultados obtenidos:

- Alta Complejidad: *Access Ratio* $\geq 1000{,}00$
- Media Complejidad: $1000{,}00 >$ *Access Ratio* $\geq 10{,}00$
- Baja Complejidad: *Access Ratio* $< 10{,}00$

Con estos criterios, se obtuvieron 7 consultas de alta complejidad, 21 consultas de media complejidad, y 41 consultas de baja complejidad. Este corpus de 69 consultas clasificadas por complejidad es utilizado para realizar los experimentos restantes.

4.4.2. Evaluación de métodos de procesado de consultas

Dado que lo que se busca evaluar es el método utilizado para procesar la consulta, y no la estructura utilizada para ello (puesto que la evaluación de la estructura utilizada se realizará en un conjunto separado de experimentos), no se tomará en cuenta la estructura utilizada para comprimir el archivo (aunque esta sí será la misma en ambos métodos), y en su lugar se evalúa el tiempo requerido para procesar un conjunto de consultas utilizando ambos métodos. Para este caso, utilizaremos la consulta con el Access Ratio más alto de cada categoría, y se evaluará el tiempo requerido para procesar dicha consulta por parte de cada método. Los tiempos obtenidos se presentan en la Tabla 9.

Tabla 9: Tiempos de respuesta a 3 consultas utilizando los métodos en estudio. Fuente: Elaboración propia.

Complejidad	Consulta	Tiempo método genérico [ms]	Tiempo método optimizado [ms]
Alta	'eso'	11469211	5862010
Media	'GALAXIES'	365234	928.52
Baja	'HISTORY'	49359.1	99.6794

Es evidente que el método optimizado es la mejor opción para el procesado de consultas, obteniendo tiempos de respuesta que son al menos un orden de magnitud menores a los obtenidos por el método genérico.

La notoria diferencia entre los tiempos de respuesta de ambos métodos se puede explicar por la dependencia del método genérico en funciones extract de la librería SDSL para obtener los nombres de columnas y datos de la estructura comprimida. Esta función, que se basa en operaciones `access` de la estructura de soporte, requiere el uso del arreglo de sufijos y de la tabla BWT del CSA para entregar el texto original entre un punto de inicio y término específicos, lo que la hace más costosa que las funciones count (utilizada para contar el número de ocurrencias) y locate (utilizada para obtener las ubicaciones de todas las ocurrencias) de la misma librería.

4.4.3. Determinación de baseline

Para determinar el espacio y tiempos del baseline, se utilizó una tabla en una base de datos PostgreSQL con 2 columnas:

- **value**, que contiene el dato correspondiente a la fila actual, y
- **column_name**, que indica el nombre de la columna a la que pertenece el dato en cuestión en el archivo original.

Estas 2 columnas representan, efectivamente, el esquema del archivo intermedio utilizado por el método optimizado que se seleccionó en el experimento anterior. Esta tabla fue llenada con datos provenientes del Archivo TAP, de forma de obtener un baseline comparable con los siguientes experimentos.

Se procesaron todas las consultas contenidas en el corpus de consultas, y se calculó el promedio de tiempos de respuesta general y por categoría de consulta, además de obtener el tamaño de la tabla en memoria para obtener su factor de espacio. Estos resultados se presentan en la Tabla 10.

Tabla 10: Resumen de tiempos de respuesta a consultas del corpus, utilizando PostgreSQL. Fuente: Elaboración propia.

Categoría	Tiempo promedio [ms]
General	313.417
Alta Complejidad	316
Media Complejidad	311.399
Baja Complejidad	314.009
Espacio: 2.0695	

Algo que es posible observar de forma inmediata es la consistencia en los tiempos de respuesta, independiente de la complejidad de la consulta utilizada.

4.4.4. Determinación de estructura óptima para compresión

Para determinar la estructura óptima para la compresión del archivo intermedio en memoria, se probarán todas las combinaciones de estructuras de soporte sobre un Compressed Suffix Array, variando además la densidad de sampleo del mismo. De esta forma, obtendremos una curva espacio v/s tiempo para cada estructura de soporte a probar.

Para efectos de poder comparar de mejor forma los resultados obtenidos en esta sección con el baseline, los resultados se desagregarán por complejidad de consulta, además de incluir el tiempo y espacio logrado por el baseline en cada categoría. Los resultados obtenidos se muestran en las Figuras 7, 8, 9 y 10.

Además de lo anterior, es interesante observar la compresión promedio (sobre todos los valores de densidad de sampleo probados) lograda por cada una de las estructuras probadas, no sólo respecto al archivo intermedio sino también respecto al archivo original, lo que corresponde a una medida de compresión más cercana a la perspectiva de un usuario del

sistema. Junto a las dos alternativas mencionadas, también se incluye el cálculo del factor de espacio sobre un archivo original que únicamente incluye los datos, quitando las descripciones de las columnas y otros tags XML innecesarios. Los promedios de espacio para todas las alternativas se presentan en la Tabla 11.

Tabla 11: Promedios de espacio para todas las estructuras probadas, calculados sobre 3 tipos de archivos de origen.

Fuente: Elaboración propia.

Estructura	Espacio promedio sobre...		
	Archivo original	Archivo original sólo datos	Archivo intermedio
wt_rlmn<>	0.2622	0.5633	0.3041
wt_rlmn<rrr_vector>	0.2361	0.5071	0.2737
wt_ap<>	1.0092	2.1679	1.1702
wt_ap<wm_int, wm_int>	1.1858	2.5471	1.3749
wt_ap<wm_int, wt_gmr>	1.5295	3.2854	1.7735
wt_ap<wt_huff, wt_huff>	1.0683	2.2948	1.2387
wm_int<>	1.2952	2.7821	1.5018
wm_int<rrr_vector>	0.2859	0.6141	0.3315
wm_int<sd_vector>	1.6492	3.5427	1.9123
wt_huff<>	1.0535	2.2630	1.2215
Baseline PostgreSQL	2.0695	4.4455	2.3997

Aquí es posible realizar algunas observaciones interesantes. En primer lugar, dado que lo único que varía entre cada gráfico es la categoría de consulta analizada, las curvas de espacio v/s tiempo conservan su forma y su posición relativa dentro del gráfico, siendo el único factor cambiante el tiempo de respuesta a una consulta. Así, es clara la diferencia entre las magnitudes de tiempo de respuesta existentes entre las diferentes categorías de consultas, lo que sugiere que la solución propuesta es sensible a la complejidad de la consulta realizada.

Otra observación importante es la diferencia que existe entre el gráfico general y los gráficos por categorías. Mientras que los Gráficos 8, 9 y 10 utilizan el promedio de los experimentos realizados para obtener la curva espacio/tiempo, el Gráfico 7 utiliza la mediana de dichos valores, de forma de evitar que las mediciones de las consultas de complejidad alta (que son varios ordenes de magnitud mayores a las de complejidad baja o media) sesguen el promedio general de cada estructura probada.

Tanto en el gráfico general como en los gráficos por complejidad, es posible observar que un grupo de estructuras logra mejores resultados que el resto de las estructuras probadas, siendo estas las únicas que logran espacios bajo 0.5. Estas estructuras, `wt_rlmn` (con y sin `rrr_vector`) y `wm_int` con `rrr_vector`, logran aprovechar de mejor forma las características del archivo intermedio para lograr mejores resultados de espacio.

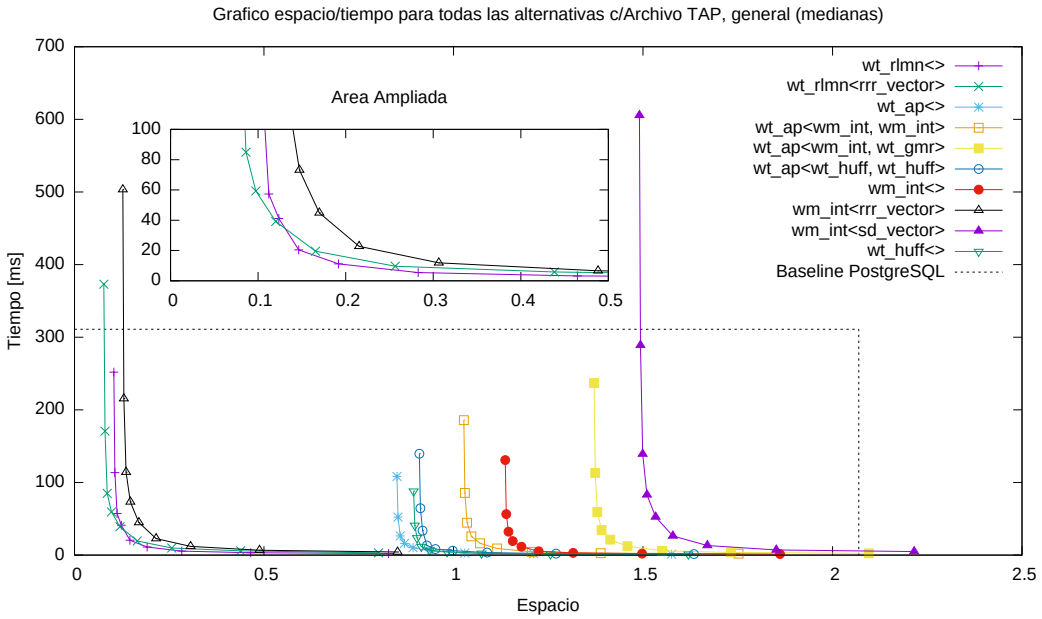


Figura 7: Resultados generales de espacio v/s tiempo para las estructuras probadas, utilizando el Archivo TAP como fuente de datos, y considerando la mediana de los valores de tiempo. Las líneas punteadas representan el espacio y tiempo logrados por el baseline, mientras que el subgráfico dentro de la figura corresponde a un acercamiento en la región de espacio hasta 0.5.

Fuente: Elaboración propia.

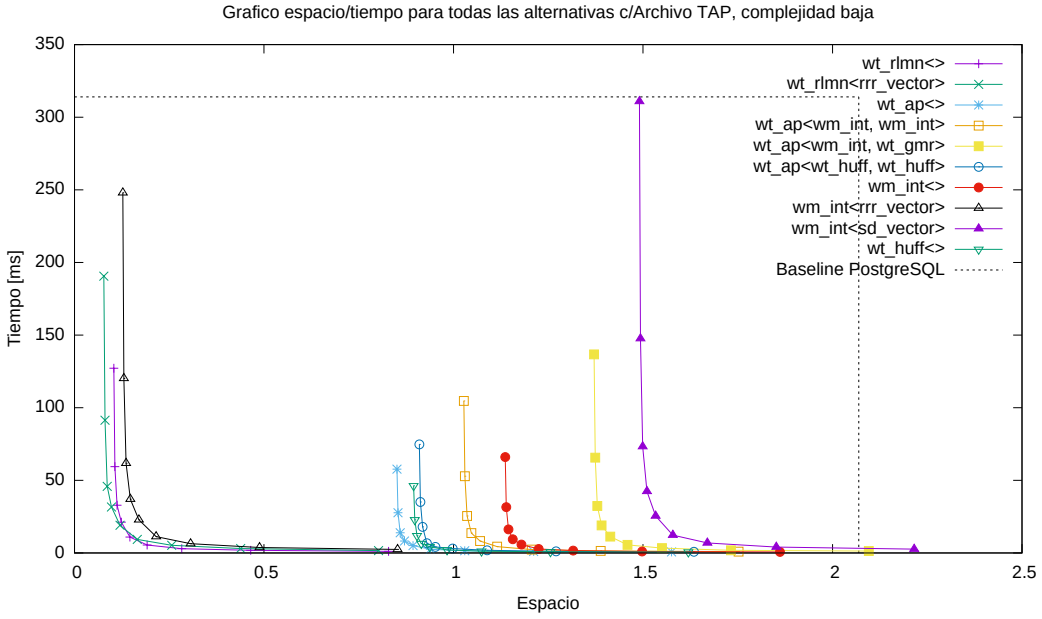


Figura 8: Resultados de espacio v/s tiempo para consultas de complejidad baja, utilizando el Archivo TAP como fuente de datos. Las líneas punteadas representan el espacio y tiempo logrados por el baseline.

Fuente: Elaboración propia.

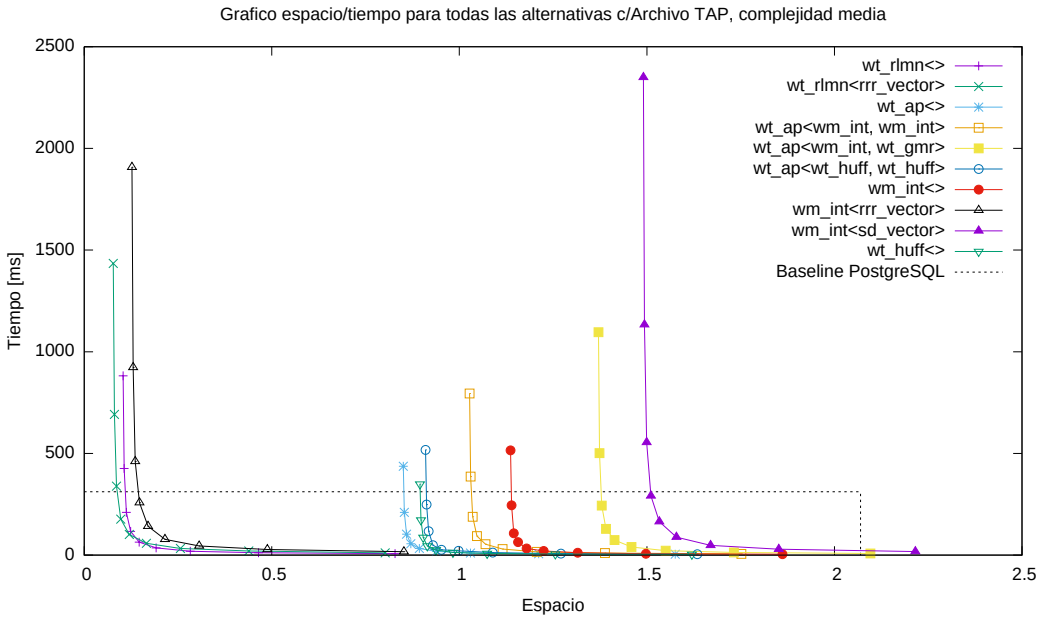


Figura 9: Resultados de espacio v/s tiempo para consultas de complejidad media, utilizando el Archivo TAP como fuente de datos. Las líneas punteadas representan el espacio y tiempo logrados por el baseline.

Fuente: Elaboración propia.

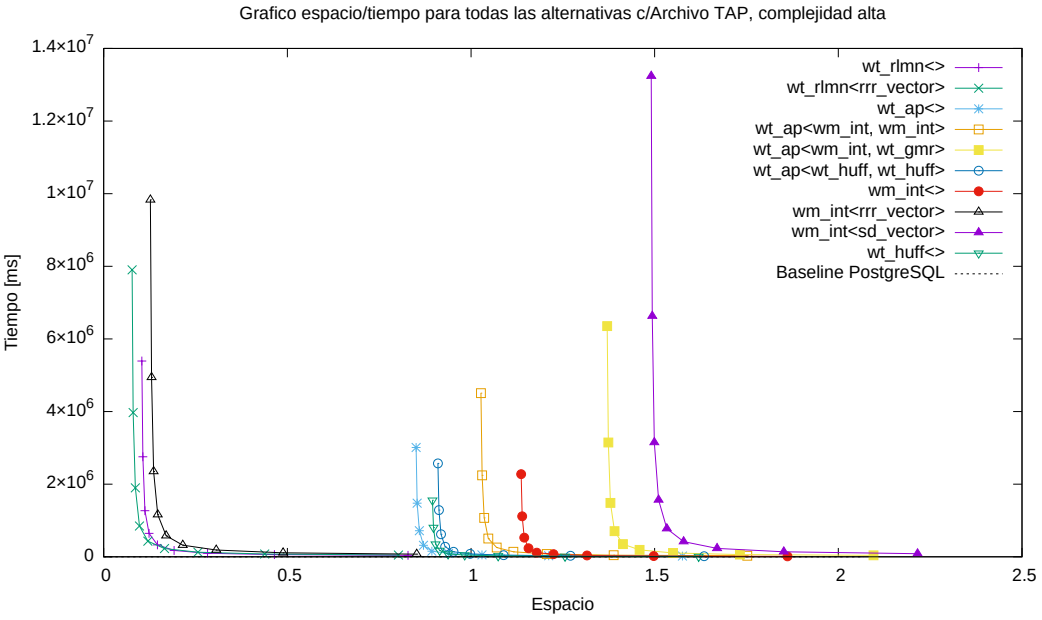


Figura 10: Resultados de espacio v/s tiempo para consultas de complejidad alta, utilizando el Archivo TAP como fuente de datos. Las líneas punteadas representan el espacio y tiempo logrados por el baseline, las que en esta figura están muy cercanas al cero y no son fácilmente visibles debido a la escala utilizada.

Fuente: Elaboración propia.

Además de los gráficos de espacio v/s tiempo mostrados anteriormente, se obtuvo el uso de memoria para la creación de algunas de las estructuras probadas en esta sección. Dichos gráficos de uso de memoria se muestran en las Figuras 11, 12, 13 y 14. Es importante notar que, en todos los gráficos, el peak de uso de memoria corresponde a 580MB, dado que el uso de memoria durante la construcción de la estructura depende en mayor medida del tamaño del documento a comprimir más que en la estructura de soporte escogida o sus parámetros.

En todos estos gráficos, los colores representan distintos procesos de la creación del CSA, los cuales cambian dependiendo de la estructura de soporte utilizada en cada caso. En todos los casos, el color rojo representa la creación del arreglo de sufijos que, coincidentemente, es el proceso con el mayor consumo de memoria entre todos los procesos.

4.4.5. Aplicación práctica: Prototipo de buscador

En la Propuesta de Solución, se propuso una arquitectura distribuida que separaba los procesos de indexación (esto es, la creación de los índices comprimidos a partir de los archivos XML VOTable obtenidos de los servicios TAP) del proceso de búsqueda. En esta sección, se presentarán algunos resultados emanados del desarrollo y pruebas de esta arquitectura, en la forma de un prototipo.

Como se mencionó en el capítulo anterior, la implementación de este prototipo fue realizada en Python y en C++, con el objetivo de aprovechar las características únicas de cada lenguaje en la implementación del prototipo. Las secciones desarrolladas en C++ (indexador y buscador) son las secciones más críticas a la hora de asegurar un buen desempeño en tiempo y espacio en el prototipo, mientras que el rastreador y despachador fueron desarrollados en Python, por su facilidad de utilizar recursos web y de manejar archivos y multiples hilos de ejecución de forma sencilla, además de proveer compatibilidad multiplataforma. Al respecto, es importante mencionar que el indexador cuenta con una sección que fue desarrollada en Python, la que se encarga de realizar la coordinación entre los distintos hilos de trabajo, mientras que el componente que realiza la indexación de un archivo XML VOTable y posterior compresión fue desarrollado en C++ utilizando la SDSL.

Tanto el indexador como el despachador siguen una arquitectura distribuida, en la cual se crean hilos (o *threads*) de trabajo para realizar los procesos de indexación y búsqueda de forma más eficiente, aprovechando los múltiples núcleos de procesamiento presentes en los servidores modernos para paralelizar estas tareas. La principal ventaja de este enfoque es la posibilidad de trabajar con datasets con cientos de archivos eficientemente, como el Dataset TAP definido en la Sección 4.2. Para asegurarnos de aprovechar al máximo los recursos de la máquina utilizada, se crean tantos *threads* como núcleos de procesamiento posea la máquina.

Bajo esta arquitectura, el proceso de indexado toma entre 25 y 30 minutos en procesar los 242 archivos que componen el Dataset TAP, utilizando los 16 cores de una máquina con un

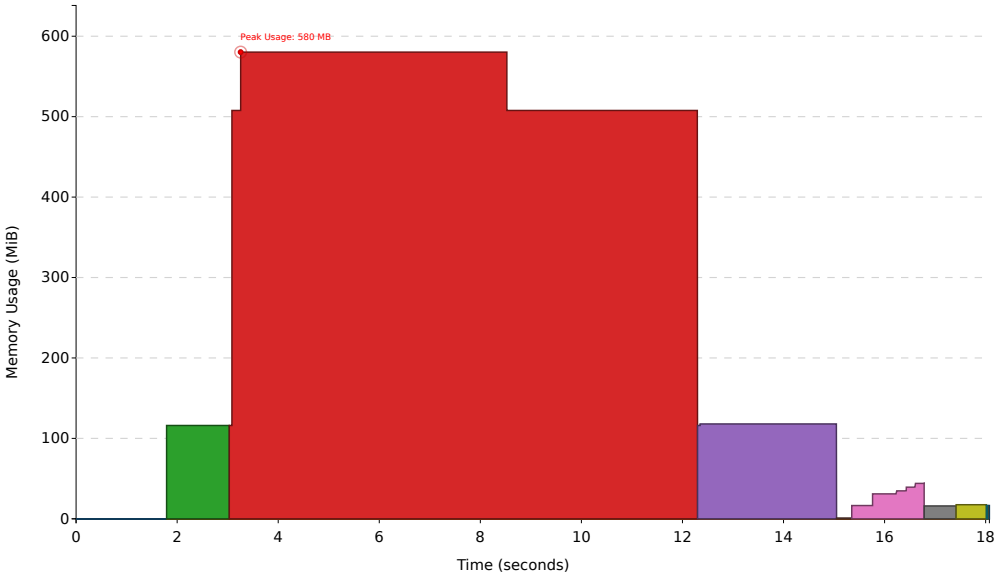


Figura 11: Uso de memoria para la creación de CSA basado en Archivo TAP, utilizando wt_rlmn y densidad de sampleo 256, con un peak de 580MB. Significados de colores, de izquierda a derecha: Parseo de texto de entrada, construcción SA, construcción BWT, construcción WT, sampleo SA, sampleo ISA.

Fuente: Elaboración propia.

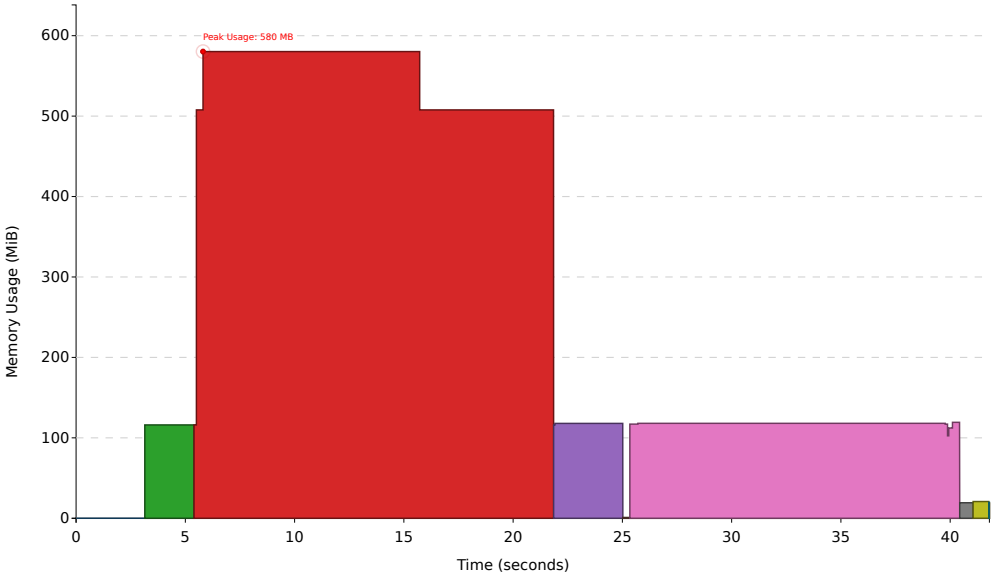


Figura 12: Uso de memoria para la creación de CSA basado en Archivo TAP, utilizando wm_int<rrr_vector> y densidad de sampleo 256, con un peak de 580MB. Significados de colores, de izquierda a derecha: Parseo de texto de entrada, construcción SA, construcción BWT, construcción WT, sampleo SA, sampleo ISA.

Fuente: Elaboración propia.

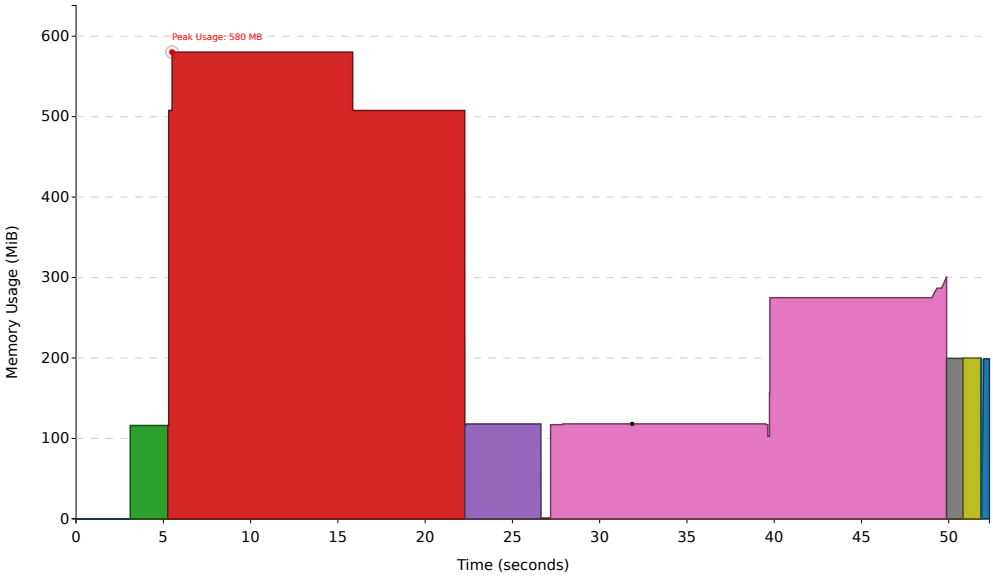


Figura 13: Uso de memoria para la creación de CSA basado en Archivo TAP, utilizando wm_int<sd_vector> y densidad de sampleo 1024, con un peak de 580MB. Significados de colores, de izquierda a derecha: Parseo de texto de entrada, construcción SA, construcción BWT, construcción WT, sampleo SA, sampleo ISA.

Fuente: Elaboración propia.

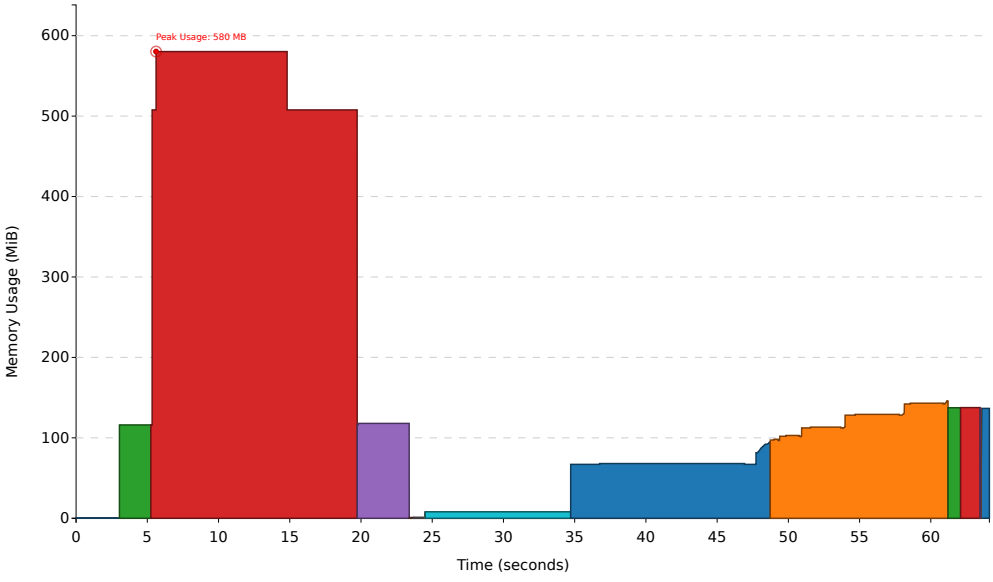


Figura 14: Uso de memoria para la creación de CSA basado en Archivo TAP, utilizando wt_ap<wm_int,wm_int> y densidad de sampleo 2048, con un peak de 580MB. Significados de colores, de izquierda a derecha: Parseo de texto de entrada, construcción SA, construcción BWT, construcción WT, class WT, offset WT's, sampleo SA, sampleo ISA.

Fuente: Elaboración propia.

Intel(R) Xeon(R) CPU E5-2630 v3 y 128GB de RAM. A pesar de que se utilizó una máquina con una alta cantidad de memoria RAM, el proceso de indexación llegó a utilizar 39.8 GB del archivo de paginación de la máquina, lo que indica que los 128GB de RAM física disponibles no fueron suficientes para realizar el proceso de indexado, aún cuando se está utilizando una arquitectura distribuida basada en *threads*. Tal como en el punto anterior, se pueden observar los gráficos de memoria para algunos de los archivos XML indexados en las figuras 15, 16, 17 y 18, los cuales colaboran en ilustrar la relación que existe entre el tamaño del archivo indexado y la memoria RAM necesaria para su indexación. Este elevado uso de memoria requerido para la indexación de los archivos más grandes del Dataset TAP, sumado al hecho de que estos procesos son ejecutados en paralelo, contribuye a que el proceso de indexación general requiera grandes cantidades de memoria RAM para ser ejecutado. Sin embargo, y a pesar de lo anterior, este procedimiento no se ejecuta con frecuencia, pudiendo ser ejecutado cada 1 o 2 semanas, y por lo tanto no representa una preocupación mayor para el desempeño de las operaciones de búsqueda sobre los documentos comprimidos.

Las operaciones de búsqueda, coordinadas por el despachador, también siguen una arquitectura distribuida basadas en *threads*. En este caso, cada *thread* tiene a su cargo la tarea de buscar sobre un subset de índice comprimidos, los cuales fueron creados en la etapa de indexación que fue comentada en el párrafo anterior. Este enfoque permite acelerar el proceso de búsqueda y aprovechar de mejor forma los recursos que posee la máquina: dado que es necesario cargar el índice comprimido en memoria y buscar sobre el mismo, y considerando la alta cantidad de archivos que podrían estar presentes en un dataset, la paralelización de este proceso ayuda a reducir los tiempos de búsqueda significativamente.

Para comprobar el funcionamiento del prototipo, se probaron algunas consultas del corpus de consultas obtenido en este capítulo. Para consultas de baja complejidad se observó un tiempo promedio de búsqueda de entre 3 y 4 segundos, mientras que para consultas de complejidad media se observaron tiempos promedios de búsqueda de entre 15 y 16 segundos, los cuales fueron medidos desde el inicio del primer *thread* al término del último *thread*, lo que contrasta con lo observado en el Baseline PostgreSQL introducido anteriormente en este capítulo. Sin embargo, es importante considerar que este Baseline corresponde a los datos de un solo documento, mientras que en este prototipo se trabaja con un dataset de 242 documentos, cada uno de los cuales debe ser buscado para construir la respuesta a la consulta del usuario. Dado que el método optimizado presentado en este documento presenta diferencias dependiendo de la complejidad de la consulta, como se ve en la Tabla 9, es de esperarse que el uso de consultas de complejidad alta signifique un aumento en los tiempos de respuesta del sistema, lo que implica que nuestro prototipo también tendrá diferencias en sus tiempos de respuesta dependiendo de la complejidad de la consulta del usuario.

Independiente de lo anterior, este prototipo demuestra el potencial de la solución planteada en una aplicación práctica y concreta, y sirve para ilustrar los beneficios de una arquitectura distribuida para aplicaciones como la planteada en este trabajo.

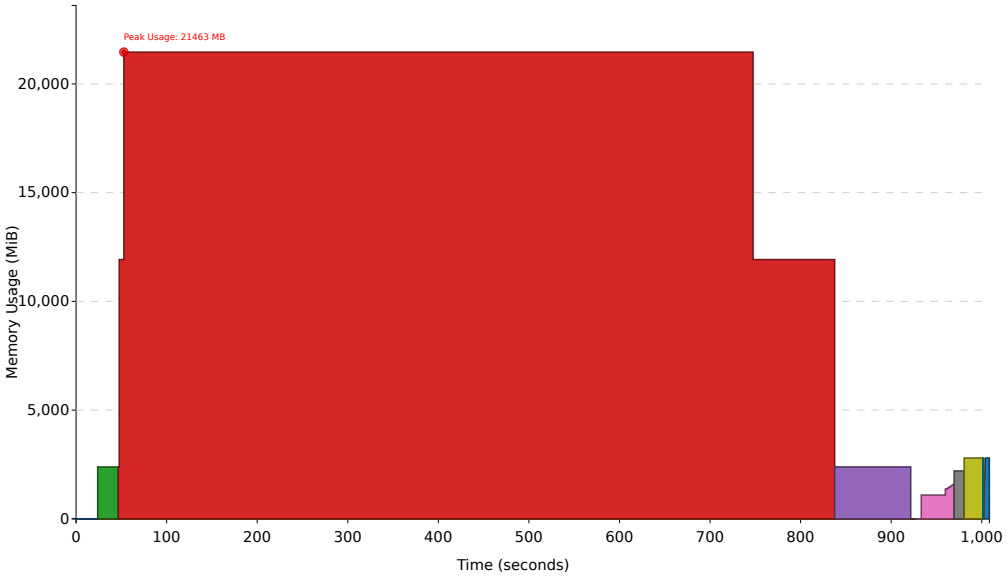


Figura 15: Uso de memoria para la indexación de un archivo de 2.54 GB, proveniente del Dataset TAP, con un peak de 21.46 GB. Significados de colores, de izquierda a derecha: Parseo de texto de entrada, construcción SA, construcción BWT, construcción WT, sampleo SA, sampleo ISA.

Fuente: Elaboración propia.

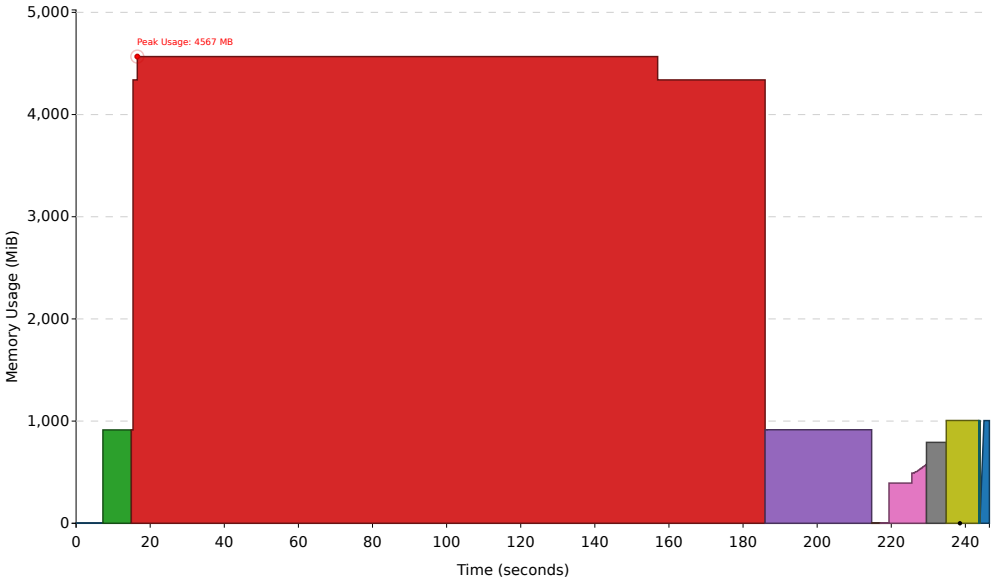


Figura 16: Uso de memoria para la indexación de un archivo de 972 MB, proveniente del Dataset TAP, con un peak de 4.57 GB. Significados de colores, de izquierda a derecha: Parseo de texto de entrada, construcción SA, construcción BWT, construcción WT, sampleo SA, sampleo ISA.

Fuente: Elaboración propia.

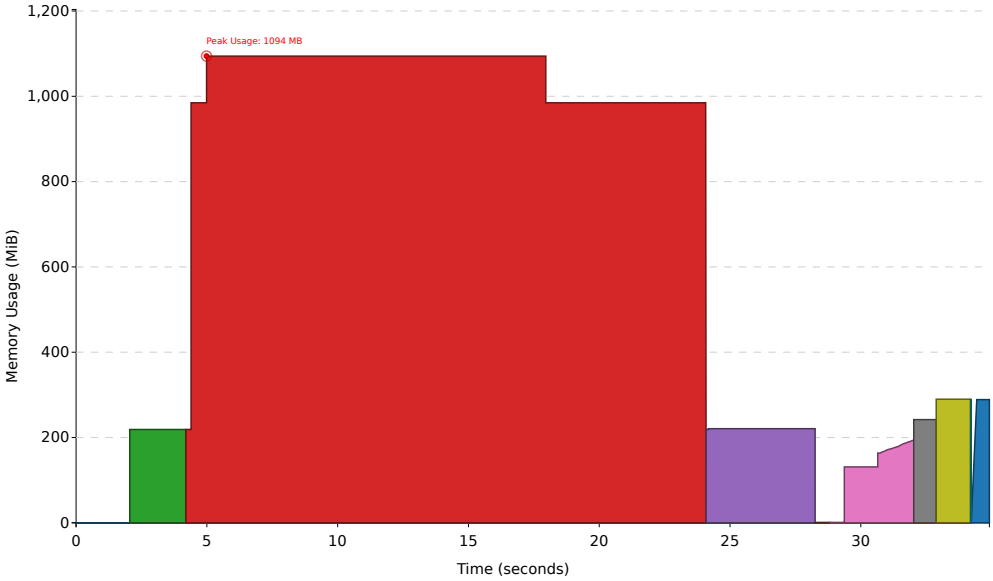


Figura 17: Uso de memoria para la indexación de un archivo de 452 MB, proveniente del Dataset TAP, con un peak de 1.09 GB. Significados de colores, de izquierda a derecha: Parseo de texto de entrada, construcción SA, construcción BWT, construcción WT, sampleo SA, sampleo ISA.

Fuente: Elaboración propia.

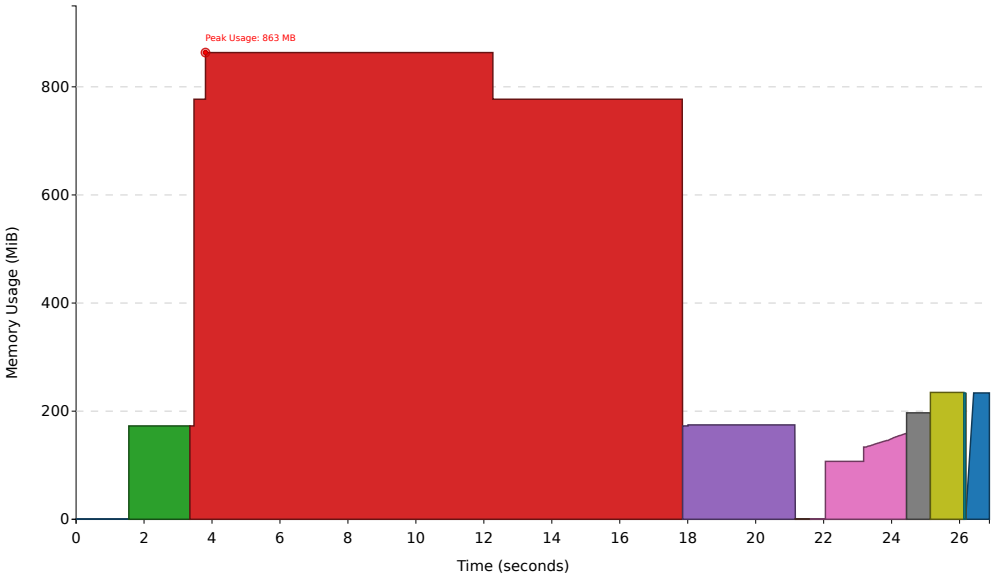


Figura 18: Uso de memoria para la indexación de un archivo de 233 MB, proveniente del Dataset TAP, con un peak de 863 MB. Significados de colores, de izquierda a derecha: Parseo de texto de entrada, construcción SA, construcción BWT, construcción WT, sampleo SA, sampleo ISA.

Fuente: Elaboración propia.

CAPÍTULO 5
CONCLUSIONES

La solución propuesta para la compresión y consulta de recursos del Observatorio Virtual resulta ser competitiva para un gran número de consultas contenidas en el corpus. Tanto para consultas de baja y media complejidad, las cuales representan el 89.95 % de las consultas presentes en el corpus, existe al menos una combinación de estructura y parámetros que permite lograr resultados de tiempo y espacio superiores a los logrados por el baseline PostgreSQL. En general, la solución propuesta logra ser 1,24 veces más rápida y 19,95 veces más pequeña que el baseline PostgreSQL (utilizando `wt_rlmn` con densidad de sampleo 2048), pudiendo lograrse mejores resultados en tiempo y/o en espacio variando la estructura de soporte utilizada y su densidad de sampleo.

Sin embargo, a pesar de que la solución propuesta en este trabajo es competitiva en un gran número de los casos analizados, el enfoque propuesto es sensible a la complejidad de la consulta realizada, a diferencia de lo mostrado por el baseline PostgreSQL. Esto queda de manifiesto al observar las diferencias de tiempo entre los Gráficos 8, 9 y 10, en donde se puede apreciar que existe una relación directa entre la complejidad de la consulta utilizada y los tiempos de respuesta logrados por la solución propuesta.

Además de lo anterior, la determinación de la complejidad de una consulta no se encuentra determinada únicamente por su frecuencia dentro del documento, sino que además depende del contexto en el que se encuentra dentro del mismo. En este sentido, una palabra que tiene una alta frecuencia dentro del documento puede corresponder a una consulta de baja complejidad, debido a que las ocurrencias de la palabra en el documento ocurren en secuencias (o *runs*) contiguas, lo que reduce considerablemente la cantidad de operaciones *access* sobre el documento original. Para medir de forma cuantitativa la complejidad de una consulta se definió el *Access Ratio*, una razón entre el número de columnas que contienen a la consulta y la cantidad de operaciones *access* sobre el documento, la que permite la categorización de las consultas utilizadas en este trabajo y, hasta donde saben los autores de este trabajo, no se encuentra presente en la literatura consultada.

Otro factor importante que incide en el tiempo de respuesta y espacio logrados por la solución propuesta es la estructura de soporte utilizada en conjunto con la CSA para comprimir los documentos en memoria. En base a los experimentos realizados en los Gráficos 7, 8, 9 y 10, se puede concluir que, en general, el uso de `wt_rlmn`, tanto con o sin `rrr_vector`, logran los mejores resultados generales tanto en tiempo como en espacio. Por otro lado, la utilización de `wt_huff` logra los tiempos más bajos de todas las estructuras probadas, incluso en las densidades de sampleo más altas. Sin embargo, dado que el enfoque propuesto en este trabajo es sensible a la complejidad de la consulta utilizada, no es posible concluir una estructura definitiva (a una densidad de sampleo dada) para su uso en consultas de cualquier complejidad, dado que una estructura que tiene un buen desempeño en consultas de cierta

dificultad podría tener un mal desempeño en consultas con una dificultad distinta. Considerando lo anterior, observando los Gráficos 8, 9 y 10, se puede concluir que el uso de `wt_rlmn`, tanto con o sin `rrr_vector`, logran los mejores resultados en términos de espacio, y logran mejores resultados que los observados en el baseline PostgreSQL para densidades de sampleo menores a 512. Al igual que en el caso general, el uso de `wt_huff` logra los tiempos más bajos de todas las estructuras probadas, incluso en las densidades de sampleo más altas (excepto en el caso de consultas de densidad media, en donde una densidad de sampleo de 2048 logra un peor resultado que el baseline PostgreSQL). Para el caso de consultas de densidad alta se mantiene la tendencia vista en las categorías de dificultad anteriormente analizadas, con la diferencia que ninguna de las estructuras analizadas, en ninguna densidad de sampleo, logra mejorar los resultados obtenidos por el baseline PostgreSQL.

Además de los resultados logrados en espacio y tiempo, es importante considerar los requisitos de memoria necesarios para la construcción de la CSA. De las Figuras 11, 12, 13 y 14, es posible observar que el peak de utilización de memoria para la construcción de la CSA en base al Archivo TAP es de 580MB, lo que es 4.11 veces el tamaño original del Archivo TAP, de 141MB. Es de esperar, entonces, que archivos más grandes requieran cantidades aún mayores de memoria RAM, tal como es posible evidenciar en las Figuras 15 y 16, en donde se observa una relación entre tamaños similar a la observada en el caso del Archivo TAP. Esto significa que, para indexación de archivos grandes, es necesario contar con grandes cantidades de memoria RAM si se desea obtener el máximo rendimiento evitando accesos a disco, lo que podría limitar el tamaño máximo de archivos indexables en algunos sistemas.

Durante el desarrollo de los experimentos y el prototipo, el uso de la librería SDSL fue fundamental para lograr un rápido ciclo de desarrollo y experimentación, gracias a que la librería SDSL cuenta con implementaciones de todas las estructuras sucintas utilizadas en la realización de este trabajo. A pesar de que la documentación es altamente técnica, y muchas veces la única documentación existente es el código mismo, las librerías y funciones de la SDSL fueron sencillas de utilizar y posibilitaron la utilización de las estructuras probadas sin la preocupación de la implementación de las mismas. Además, la utilización de una implementación estandarizada de las estructuras probadas posibilita a la reproducción de los resultados obtenidos en este trabajo.

El enfoque y arquitectura propuestos en este trabajo, pueden ser utilizados para soportar búsquedas de forma eficiente sobre recursos del Observatorio Virtual, logrando tiempos de búsqueda y espacio competitivos en comparación a lo mostrado por el baseline PostgreSQL. Sistemas como el propuesto, en donde el procesado de una consulta toma sólo un par de milisegundos (en consultas de baja y media complejidad), son altamente deseables en contextos donde es necesario proveer de un servicio a una cantidad moderadamente alta de usuarios, aún cuando para consultas de alta complejidad no se logre mejorar el baseline PostgreSQL. Sin embargo, y considerando la baja cantidad de consultas de alta complejidad en nuestro corpus (las consultas de baja y media complejidad representan un 89.95 % del corpus), es posible concluir que, en general, la solución propuesta presenta una alternativa competitiva al uso de PostgreSQL para búsqueda y descubrimiento de recursos del Observa-

torio Virtual.

Como trabajo futuro, se propone la evaluación de nuevas formas de procesamiento de consultas con estructuras sucintas, que den garantías de peor caso frente a las consultas procesadas. Esto permitiría mitigar la sensibilidad del método frente a la complejidad de la consulta utilizada, además de brindar tiempos de respuesta más consistentes y predecibles. Junto a lo anterior, se propone la implementación del prototipo creado como un servicio web, haciendo que la solución propuesta en este trabajo se encuentre disponible a la comunidad académica y científica del país, a través del Observatorio Virtual Chileno.

REFERENCIAS BIBLIOGRÁFICAS

[Abouelhoda *et al.*, 2004] Abouelhoda, M. I., Kurtz, S., y Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53 – 86. The 9th International Symposium on String Processing and Information Retrieval.

[Araya *et al.*, 2015] Araya, M., Solar, M., y Antognini, J. (2015). A brief survey on the virtual observatory. *New Astronomy*, 39:46–54.

[Arroyuelo *et al.*, 2012] Arroyuelo, D., González, S., Marín, M., Oyarzún, M., y Suel, T. (2012). To index or not to index: time-space trade-offs in search engines with positional ranking functions. En *Proc. 35th International ACM SIGIR conference on research and development in Information Retrieval (SIGIR'12)*, pp. 255–264.

[Barbay *et al.*, 2014] Barbay, J., Claude, F., Gagie, T., Navarro, G., y Nekrich, Y. (2014). Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268.

[Benson *et al.*, 2009] Benson, Kevin and Plante, Ray and Auden, Elizabeth and Graham, Matthew and Greene, Gretchen and Hill, Martin and Linde, Tony and Morris, Dave and O'Mullane, Wil and Rixon, Guy and others (2009). Ivoa registry interfaces version 1.0. *IVOA Recommendation. URL: http://www. ivoa. net/documents/RegistryInterface*.

[Bobichon y Bijaoui, 1997] Bobichon, Y. y Bijaoui, A. (1997). A regularized image restoration algorithm for lossy compression in astronomy. *Experimental Astronomy*, 7(3):239–255.

[Claude *et al.*, 2015] Claude, F., Navarro, G., y Ordóñez, A. (2015). The wavelet matrix. *Inf. Syst.*, 47(C):15–32.

[Demleitner *et al.*, 2014] Demleitner, M., Neves, M., Rothmaier, F., y Wambsganss, J. (2014). Virtual observatory publishing with dachs. *Astronomy and Computing*, 7-8:27 – 36. Special Issue on The Virtual Observatory: I.

[Dowler *et al.*, 2015] Dowler, P., Bonnarel, F., y Tody, D. (2015). Ivoa simple image access version 2.0. *IVOA Recommendation*, 23.

[Dowler *et al.*, 2010] Dowler, Patrick and Rixon, Guy and Tody, Doug and Andrews, K and Good, J and Hanisch, R and Lemson, G and McGlynn, T and Noddle, K and Ochsenbein, F and others (2010). Table access protocol version 1.0. *IVOA Recommendation*, 27.

[Ferragina *et al.*, 2009] Ferragina, P., Giancarlo, R., y Manzini, G. (2009). The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849 – 866.

[Gog *et al.*, 2014] Gog, S., Beller, T., Moffat, A., y Petri, M. (2014). From theory to practice: Plug and play with succinct data structures. En *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pp. 326–337.

[Golynski *et al.*, 2006] Golynski, A., Munro, J. I., y Rao, S. S. (2006). Rank/select operations on large alphabets: A tool for text indexing. En *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pp. 368–373, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Grossi, 2011] Grossi, R. (2011). A quick tour on suffix arrays and compressed suffix arrays. *Theoretical Computer Science*, 412(27):2964 – 2973. Combinatorics on Words (WORDS 2009).

[Grossi *et al.*, 2003] Grossi, R., Gupta, A., y Vitter, J. S. (2003). High-order entropy-compressed text indexes. En *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pp. 841–850, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Hanisch *et al.*, 2001] Hanisch, R. J., Farris, A., Greisen, E. W., Pence, W. D., Schlesinger, B. M., Teuben, P. J., Thompson, R. W., y Warnock, A. (2001). Definition of the flexible image transport system (fits). *Astronomy & Astrophysics*, 376(1):359–380.

[Jacobson, 1989] Jacobson, G. (1989). Space-efficient static trees and graphs. En *30th Annual Symposium on Foundations of Computer Science*, pp. 549–554.

[Mäkinen y Navarro, 2005] Mäkinen, V. y Navarro, G. (2005). Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66.

[Manber y Myers, 1990] Manber, U. y Myers, G. (1990). Suffix arrays: A new method for on-line string searches. En *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pp. 319–327, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Ochsenbein *et al.*, 2011] Ochsenbein, François and Williams, Roy and Davenhall, Clive and Durand, Daniel and Fernique, Pierre and Giaretta, David and Hanisch, Robert and McGlynn, Tom and Szalay, Alex and Taylor, Mark B and others (2011). Votable format definition version 1.3. *IVOA Recommendation*.

[Okanohara y Sadakane, 2007] Okanohara, D. y Sadakane, K. (2007). Practical entropy-compressed rank/select dictionary. En *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pp. 60–70, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Ortiz *et al.*, 2006] Ortiz, I., Lusted, J., Dowler, P., Szalay, A., Shirasaki, Y., Santisteban, M. O., O'Mullane, W., y Osuna, P. (2006). Ivoa astronomical data query language version 2.0. *IVOA Recommendation*, 20081030.

[Raman *et al.*, 2007] Raman, R., Raman, V., y Satti, S. R. (2007). Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4).

[Reghbati, 1981] Reghbati, H. K. (1981). Special feature an overview of data compression techniques. *Computer*, (4):71–75.

[Shanmugasundaram y Lourdusamy, 2011] Shanmugasundaram, S. y Lourdusamy, R. (2011). A comparative study of text compression algorithms. *International Journal of Wisdom Based Computing*, 1(3):68–76.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.

[Solar *et al.*, 2015] Solar, M., Araya, M., Arévalo, L., Parada, V., Contreras, R., y Mardones, D. (2015). Chilean virtual observatory. En *2015 Latin American Computing Conference (CLEI)*, pp. 1–7.

[Tody *et al.*, 2012] Tody, Doug and Dolensky, Markus and McDowell, Jonathan and Bonnarel, Francois and Budavari, Tamas and Busko, Ivo and Micol, Alberto and Osuna, Pedro and Salgado, Jesus and Skoda, Petr and others (2012). Simple spectral access protocol version 1.1. *IVOA Recommendation*, 10.

[Vohl *et al.*, 2015] Vohl, D., Fluke, C., y Vernardos, G. (2015). Data compression in the petascale astronomy era: A gerlumph case study. *Astronomy and Computing*, 12:200 – 211.

[Williams *et al.*, 2008] Williams, R., Hanisch, R., Szalay, A., y Plante, R. (2008). Simple cone search version 1.03. *IVOA Data Access Layer WG Reccommendation*.