
**Universidad Nacional Autónoma de México**
Posgrado en Ciencia e Ingeniería de la Computación

# Optimización de agregaciones extendidas utilizando SQL en SMBD objeto relacionales, columnares y de arreglos.

**TESIS**
que para optar por el grado de:
maestro en ciencia e ingeniería de la computación

presenta:
anuar cadena rico

Director de Tesis:
Dr. Javier García García
Centro de Ciencias de la Complejidad

Ciudad Universitaria, CDMX Noviembre del 2017.

# ÍNDICE GENERAL

# ÍNDICE DE TABLAS

# ÍNDICE DE FIGURAS

# CAPÍTULO 1

# Introducción

Este tipo de problemas de integridad, en particular de integridad referencial, hoy en día han resurgido dado que se presentan en bases de datos no solamente relacionales, también se presenta en bases de datos no estructuradas donde esté presente algún tipo de referencia. Debido a la ocurrencia de este tipo de errores se ha generado un gran interés en los últimos años, la búsqueda de soluciones como las que se presentan en este trabajo.

En este trabajo se tratarán las Funciones de Agregación Extendida [1] sobre las funciones de agregación del estándar SQL [2] las cuales son las siguientes: count(*), count(), sum(), avg(), min(), max(). Estas funciones son una herramienta muy útil que permite que con base a las probabilidades de ocurrencia de valores en un conjunto de registros que tienen una referencia indefinida o inválida en su llave foránea se calculen las funciones de agregación dando como resultado su valor más probable. De ésta forma no se tiene que eliminar o modificar las tablas que contienen dichos registros con problemas de integridad referencial y así la información sigue en el estado en el que se encontró, pero es posible obtener información útil para la toma de decisiones.

Estas agregaciones extendidas [1] como se mencionó anteriormente no modifican ni la tabla referencia ni la tabla referenciada, esto da un método mucho más útil en comparación a trabajos relacionados [3] [4] [5], así también evitan utilizar métodos comúnmente usados como lo es modificar todos los registros que tienen un problema de integridad referencial, creando una referencia válida en la tabla referencia a la cual apunten todos estos registros. Este valor puede ser un valor simbólico cualquiera para denotar que éstas son referencias desconocidas.

También se puede utilizar otro método que consiste en la eliminación de todos los registros que tengan problemas de integridad referencial, para conseguir una base de datos en un estado consistente, pero esto significaría que un porcentaje considerable de datos se perdería, algo que no parece la mejor opción debido a que los datos borrados representan operaciones, que no se tomarán en cuenta, y podrían significar una mala toma de decisiones para la empresa u organización dueña de los mismos.

La otra opción es referenciar esos datos a uno de los datos válidos que existan en la tabla, esto sería conveniente para eliminar los problemas de integridad referencial y de igual forma dejar una base de datos en un estado consistente pero cuando se hagan cálculos con respecto a los datos, ese dato al que se referenció, no estaría dando un resultado correcto, esto podría ser un problema si se utilizan estos datos para la toma de decisiones.

## 1.1 Motivación

En las bases de datos relacionales sean de gran tamaño o un tamaño moderado, existe un problema muy común que ha estado presente en las mismas desde que se publicaron las bases de éstas en la década de los 70's, es el caso de los errores de integridad referencial donde uno o varios registros de una tabla hacen referencia a un elemento no existente dentro de otra tabla. Lo anterior representa un estado inconsistente de una base de datos.

Existen en el mercado opciones para reparar el problema, que o bien eliminan los registros con problemas o cambian las referencias inválidas a una válida, pero estos métodos modifican la base de datos. Esto hace que la toma de decisiones respecto a los datos resultantes de cualquiera de esos métodos, no tenga un 100% de fiabilidad, provocando problemas en mayor o menor escala según la cantidad de datos que haya tenido el problema.

El presente trabajo pretende aportar información en relación a la utilización de Funciones de Agregación Extendida [1] lo que evita el uso de cualquiera de los métodos anteriores debido a que estas funciones utilizan las probabilidades de ocurrencia, no modifican los datos de la base de datos, dan una aproximación respecto a los datos con mayor repetición en la tabla referenciada, este es un mejor enfoque en la toma de decisiones debido a que se toman en consideración los datos con problemas de integridad referencial.

Otra aportación del presente trabajo es la ejecución de las consultas mencionadas en diferentes Sistemas Manejadores de Bases de Datos, para poder ver el rendimiento de cada uno y contestar la pregunta: ¿Qué familia de SMBD o SMBD especifico es el más indicado para ejecutar Funciones de Agregación Extendida sobre tablas con problemas de integridad referencial? Se utilizarán SMBD Relacionales, SMBD Relacionales por columnas, Sistema de Computación Distribuida, SMBD por arreglos, cada uno de estos sistemas tiene un enfoque diferente en cuanto al manejo de datos.

Un punto que se tiene que tratar en el presente trabajo, es que a pesar de que las pruebas se están haciendo en SMBD muy recientes, el problema de integridad, particularmente integridad referencial, hoy en día se esta volviendo a generar gran interés dado que las grandes bases de datos, manejadas en temas de big data, tienen este tipo de problemas, y si bien el presente estudio se basa en bases de datos relacionales, los conceptos aquí tratados pueden ser utilizados en bases de datos no estructuradas para resolver sus problemas de integridad.

## 1.2 Propuesta

La propuesta es hacer un estudio comparativo de las Funciones de Agregación Extendida [1] y probarlas en una base de datos sintética, TPC-H [6], utilizando un factor de escala de 10. A esta base de datos se le introdujeron errores de integridad referencial distribuidos de manera aleatoria y con los siguientes porcentajes de errores de integridad: 5%, 10%, 15%, 20% y 30% cada uno representa una tabla referenciada para tener un margen de referencia mayor para su análisis, las consultas se ejecutaron en 4 sistemas con características y enfoques diferentes, adecuando la consulta a cada uno de los sistemas.

PostgreSQL: Éste será el Sistema Gestor de Bases de Datos Relacional en el cual la ejecución de la consulta servirá como base para posteriormente comparar los tiempos de ejecución de las Funciones de Agregación Extendida con los otros sistemas. Este sistema no requiere herramientas adicionales para la implementación de las Funciones de Agregación Extendida [1], ya que cuenta con todo lo necesario para cada función.

HP Vertica: En este Sistema Gestor de Bases de Datos Relacionales con un almacenamiento columnar, la ejecución de consultas es muy parecida a la de PostgreSQL sólo que el almacenamiento se enfoca en las columnas, aquí para la implementación de las funciones max() y min() se utilizará una herramienta adicional que es el programa Python para los cálculos necesarios, además un JDBC para la conexión entre HP Vertica con Python.

SciDB: Éste es un Sistema Manejador de Bases de Datos enfocado en arreglos, en este sistema se cambiará la consulta utilizada anteriormente de SQL al lenguaje AFL, éste es un lenguaje de manejo de arreglos, como herramienta adicional se utilizará la librería SciDBR que es la implementación de R en SciDB para poder manejar las funciones matemáticas que se necesitan en el cálculo de las funciones max() y min(), para poder conectar estas dos aplicaciones se utilizará Shim.

Apache Spark: En este sistema se utilizará un conector JDBC para conectar este sistema con el SMBD PostgreSQL así poder tomar los datos de las tablas y guardarlos en DataFrames que posteriormente se guardarán de forma permanente en archivos .parquet, para su utilización cuando sean necesarios y después se ejecutarán las consultas sobre ellos. De igual manera este sistema no necesita la instalación de herramientas adicionales.

## 1.3 Trabajos relacionados

En [3] se propone un marco de referencia para reparar bases de datos inconsistentes, mediante operaciones de inserción y eliminación en conjuntos de instrucciones llamados átomos. En la teoría presentada en [4] se pretende reparar la base de datos mediante una actualización por medio de inserción y eliminación de elementos o un conjunto de ambas acciones, estas actualizaciones dejarán a la base de datos en un estado consistente.

En [5] teniendo un panorama en el que, o se modificaban las bases de datos insertando o eliminando filas enteras para dejar a las bases de datos en un estado lo mas cercano a consistente, pero esa fila que se inserto o se elimino tiene componentes tanto correctos como incorrectos. Así que lo que se propone en [5] es solamente reparar, mediante actualizaciones, los componentes con errores, dejando igual los componentes libres de errores, el modelo implica remplazar los valores erróneos por variables.

Por su parte en [7] se desarrolló un navegador de calidad de datos, la función principal de esta herramienta es la limpieza de datos, la cual utilizó recopilaciones de datos para agilizarlo y guardar perfiles dentro de la base de datos. En [8] también se habla de la limpieza de datos, en las bases de datos operacionales en el proceso de integración de los datos al data warehouse, dando un panorama de los problemas y las medidas que se debe de tomar cuando se realice.

## 1.4 Objetivos

### 1.4.1 General

El objetivo del presente trabajo es experimentar con las Funciones de Agregación que se presentan en [1] en distintos sistemas manejadores de bases de datos y comparar su desempeño. Estos sistemas deben soportar el lenguaje SQL o uno equivalente para el manejo de datos. El presente estudio incluirá la experimentación de distintos sistemas manejadores de bases de datos ya sea objetos relacionales, columnares y de arreglos además de un sistema de computación distribuida.

### 1.4.2 Específicos

- Convertir las Funciones de Agregación Extendida [1] a sentencias en forma de consultas SQL para su posterior ejecución en sistemas que acepten este tipo de lenguaje o una opción similar, esto último dado que las Funciones de Agregación Extendida no son parte del estándar SQL [2].

- Escribir las Funciones de Agregación Extendida max() y min() [1] en forma de consultas SQL utilizando tablas auxiliares para su cálculo, de forma que se obtenga el mejor rendimiento. Cabe aclarar que estas variantes nunca se habían documentado.

- Cargar las tablas de la base de datos TPC-H con un factor de escala de 10 en cada uno de los 4 sistemas con un esquema relajado. El factor de escala de 10 es para que las tablas estén lo suficientemente pobladas para obtener un cálculo significativo en comparación con una base de datos real.

- Ejecutar las consultas en PostgreSQL tomando el tiempo de ejecución, manejo de memoria RAM, y porcentaje de uso de procesador al momento de ejecutar las Funciones de Agregación Extendida count(*), count(), sum(), avg() con los siguientes porcentajes de errores de integridad referencial 5%, 10%, 15%, 20% y 30%, en cuanto a las funciones max() y min() los porcentajes de errores de integridad referencial serán 5%, 10%, 15% y 20%.

- Ejecutar las consultas en HP Vertica, tomando el tiempo de ejecución, manejo de memoria RAM, y porcentaje de uso de procesador al momento de ejecutar las Funciones de Agregación Extendida count(*), count(), sum(), avg() con los siguientes porcentajes de errores de integridad referencial 5%, 10%, 15%, 20% y 30%, en cuanto a las funciones max() y min() los porcentajes de errores de integridad referencial serán 5%, 10%, 15% y 20%.

- Cambiar las consultas de SQL a una forma AFL utilizando las funciones con las que cuenta el SMBD SciDB, la consulta en presentación cambiará pero en funcionalidad será la misma, tomando el tiempo de ejecución, manejo de

memoria RAM, y porcentaje de uso de procesador al momento de ejecutar las Funciones de Agregación Extendida count(*), count(), sum(), avg() con los siguientes porcentajes de errores de integridad referencial 5%, 10%, 15%, 20% y 30%, en cuanto a las funciones max() y min() los porcentajes de errores de integridad referencial serán 5%, 10%, 15% y 20%.

- En cuanto a Apache Spark aquí se utilizará un JDBC para poder extraer las tablas necesarias para la ejecución de la consulta SQL y se guardará en DataFrames dentro de Spark que posteriormente serán pasadas a archivos con extensión .parquet, y en base a ellos se trabajara en un entorno SQL que proporciona Spark, tomando el tiempo de ejecución, manejo de memoria RAM, y porcentaje de uso de procesador al momento de ejecutar las Funciones de Agregación Extendida count(*), count(), sum(), avg() con los siguientes porcentajes de errores de integridad referencial 5%, 10%, 15%, 20% y 30%, en cuanto a las funciones max() y min() los porcentajes de errores de integridad referencial serán 5%, 10%, 15% y 20%.

- Una vez que se tengan los resultados de la ejecución de las consultas en los diferentes sistemas, se procederá a hacer un análisis comparativo, para así llegar a una conclusión sobre cuál sistema es el mejor en cuanto a la ejecución de las Funciones de Agregación Extendida: count(*), count(), sum(), avg(), max() y min().

## 1.5 Aportaciones

Se realizó un análisis con el propósito de comparar la efectividad de los 4 sistemas presentados, cada uno de éstos con un enfoque al uso de datos diferente, y con base en este análisis se obtuvo un marco de referencia de las Funciones de Agregación Extendida dentro de cada uno, esto con el fin de conocer cuál de estos sistemas tiene el mejor rendimiento respecto a las 3 principales características analizadas en los siguientes capítulos:

- Tiempo de ejecución.
- Manejo de memoria RAM.
- Uso de procesador.

Estas características se midieron en cada sistema, cuando se crearon las tablas auxiliares necesarias, como al aplicar las Funciones de Agregación Extendida, cada sistema tiene características especiales pero se realizó el análisis simulando el mismo ambiente, así el usuario que necesite utilizar Funciones de Agregación Extendida en una base de datos con errores de integridad referencial tendrá este trabajo como referencia para decidir el ambiente a utilizarla.

Este trabajo es valioso debido a que demuestra el desempeño de funciones de las agregacion importantes para el analisis de datos en distintas herramientas de explotación de datos

# CAPÍTULO 2

# Marco Teórico

El problema de información errónea entre las tablas relacionadas ha existido desde que se introdujo el concepto de las mismas, con el paso del tiempo ha habido distintos investigadores, que con base a diversas investigaciones, han propuesto métodos, teorías o formas de remediar o trabajar con estos datos erróneos.

En la propuesta basada en las Funciones de Agregación Extendida se invita a trabajar con dichos errores en vez de modificar la base de datos o eliminar dichos datos con problemas de integridad referencial, esta es una propuesta interesante para las empresas ya que ayudará en la toma de decisiones, ya que se tendrá un marco de referencia más real al que se podría tener si se ignoran, se eliminan, o se modifican los datos.

El presente capítulo esta estructurado de la siguiente forma: 2.1 Definiciones y conceptos teóricos, se definirán todos los conceptos que se utilizarán en las secciones siguientes; 2.2 Probabilidad parcial, presentará el vector de probabilidad entre los valores con referencias invalidas hacia los datos con referencias válidas y las diferentes formas de utilizarlo; 2.3 Funciones de Agregación Extendida, se presentarán las funciones y la forma de como calcularlas; 2.4 Modificación a la función min() y max(), se presentarán los cambios a las fórmulas para poder utilizarlas en los SMBD.

## 2.1 Definiciones y conceptos teóricos.

**SciDB**
Es una plataforma de software de código abierto, que puede instalarse en un servidor único, en la nube o en un conjunto de servidores. Es una plataforma de administración y análisis avanzado de datos, todo en uno. Además provee la capacidad de escalabilidad, una base de datos de siguiente generación con versionado de datos para soportar las necesidades de aplicaciones comerciales y científicas. Los modelos de arreglos proveen almacenamiento compacto de datos y un alto rendimiento de operación en datos ordenados tales como datos espaciales, datos temporales y datos

basados en matriz para operaciones de álgebra lineal.

**Arreglo**
Es una estructura que se usa en SciDB para almacenar información, consta de un número de dimensiones y atributos, que se usan como tablas, y cada celda del arreglo como filas, las dimensiones y los atributos actúan como columnas, y las dimensiones actúan como llaves primarias.

**Dimensión**
Las dimensiones en SciDB son usadas al igual que los atributos para almacenar datos individuales y sirven como llaves primarias, el tamaño de la dimensión se determina por el rango que hay entre el inicio y el final de la dimensión.

**Atributos**
Los atributos en SciDB son usados para almacenar valores de datos individuales en las celdas de un arreglo.

**Servidor único**
Es un tipo de configuración dentro de SciDB que hace referencia a una sola computadora que puede tener múltiples procesadores, memoria y almacenamiento interno.

**Iquery**
Es la herramienta de comando de línea básica para comunicarse con SciDB, es el cliente básico para usar comandos AQL y AFL.

**AQL**
Array Query Language. Es un lenguaje declarativo de alto nivel para trabajar con arreglos dentro de SciDB, es muy parecido al lenguaje SQL para bases de datos relacionales pero usa un modelo de datos basados en arreglos.

**AFL**
Array functional Language es un lenguaje funcional para trabajar con arreglos dentro de SciDB. Los operadores en este lenguaje son usados para hacer consultas o declaraciones.

**R**
Es un lenguaje y un entorno para computación estadística y gráfica. Provee una amplia variedad de modelos estadísticos, además de ser ampliamente extensible.

Es un conjunto de software integrado para facilitar la manipulación, cálculo y visualización gráfica de datos. Incluye un tratamiento y almacenamiento eficaz de datos. Además de una serie de operadores para cálculos sobre arreglos, en particular de matrices. También tiene la característica de tener una gran colección coherente e integrada de herramientas para el análisis de datos, e instalaciones gráficas para el

análisis y visualización de datos en pantalla o papel. Es un lenguaje de programación bien desarrollado, sencillo y efectivo que incluye condicionales, bucles, funciones recursivas definidas por el usuario y, recursos de entrada y salida Es un software libre bajo los términos de la licencia al público en general de GNU en código abierto.

**SciDBR**

Es un paquete de R que ayuda a interactuar con SciDB, ejecutando directamente las consultas utilizando el lenguaje nativo AFL, con la opción de regresar los resultados como data frames.

**Dataframe**

Es una estructura parecida a un arreglo bidimensional, en la que cada columna contiene mediciones de una variable y cada fila contiene un caso. Es una lista de vectores de igual tamaño, usado para almacenar tablas de datos.

**Shim**

Es un servicio web que expone una API muy simple para que los clientes interactúen con las conexiones de SciDB sobre HTTP. Este consiste en un pequeño número de servicios. Los clientes de ésta API comienzan pidiendo un ID de sesión de servicio para posteriormente ejecutar la consulta y liberar el ID de sesión, estos ID son distintos de los de consulta de SciDB, un ID de sesión Shin agrupa una consulta Scidb junto con el servidor para la entrada y salida del cliente.

**HP Vertica**

En un manejador de base de datos analítico, y está diseñado para su uso en almacenes de datos y en otros lugares donde haya grandes cargas de trabajo de datos en donde la velocidad, escalabilidad, sencillez y la apertura, son temas cruciales. Vertica se basa en una arquitectura distribuida y en una compresión columnar para ofrecer una velocidad mayor. Se puede utilizar en instalaciones locales, en la nube o en hadoop. Soporta consultas en tiempo real y se pueden cargar los datos hasta 10 veces más rápido en comparación al almacenamiento en filas ya que usa un modelo transaccional de viaje en el tiempo único que asegura una concurrencia extremadamente alta de consultas mientras se cargan nuevos datos al sistema a la vez.

**Almacenamiento columnar**

Ofrece ganancias significativas en el rendimiento, entrada/salida, almacenamiento y eficiencia cuando se trata de analizar cargas de trabajo. Con el almacenamiento en columnas las consultas solo leen las columnas necesarias para responder la consulta.

**PostgreSQL**

Es un sistema de gestión de bases de datos relacional, de código abierto. Utiliza un

modelo cliente servidor y usa multiproceso en vez de multihilos para garantizar la estabilidad del sistema. Un fallo en uno de los procesos no afectará el resto y el sistema continuará funcionando. Se caracteriza por su estabilidad, potencia, robustez, facilidad de administración e implementación de estándares. Además de funcionar muy bien con grandes volúmenes de datos y una alta concurrencia de usuarios accediendo a la vez en el sistema.

**Apache Spark**
Es un sistema de computación de cluster, rápido y de propósito general, el cual proporciona APIs de alto nivel en java, Scala, Pthon y R, ademas de un motor optimizado que soporta gráficos de ejecución general. También soporta un gran conjunto de herramientas de alto nivel, como Spark SQL y procesamiento de datos estructurado, Mlib para el aprendizaje automático, GraphX para procesamiento de gráficos y Spark Streaming.

**Spark SQL**
Es un módulo Spark para el procesamiento estructurado de datos, proporcionan información sobre la estructura de los datos y el cálculo que se realiza internamente, se puede interactuar con este mediante SQL y API Dataset, los resultados además se ejecutan utilizando el mismo motor de ejecución, esto es independientemente de que API o lenguaje se haya utilizado para expresar el cálculo.

**Spark JDBC**
Spark SQL también incluye una fuente de datos que pueden leer datos desde otras bases de datos utilizando JDBC, esta función regresa los resultados en formato DataFrame y pueden ser fácilmente procesados en Spark SQL o unidos con otros data sources. La fuente de datos JDBC es también fácil de usar desde java o Pyton, no requiere que el usuario provea un ClassTag.

**TPCH**
Consiste en un conjunto de consultas ad-hoc orientadas al negocio y modificaciones de datos concurrentes. Las consultas y los datos que pueblan la base de datos han sido elegidos para tener una amplia relevancia en toda la industria, manteniendo al mismo tiempo un grado suficiente de facilidad de implementación

**Notación combinatoria**
Dado un conjunto de $n$ elementos distinguibles, se le llama combinación sin repetición de $p$ elementos con $p<n$, elegidos entre los $n$ elementos, a cualquier subconjunto de $p$ elementos distintos del conjunto. El número de combinaciones sin repetición de $p$ elementos elegidos entre los $n$ se nota habitualmente:

$$\binom{n}{p} = \frac{n!}{p!(n-p)!} \tag{2.1}$$

## 2.2 Probabilidad parcial.

En cuanto a la probabilidad parcial presentada en [9], la cual se refiere a un vector de probabilidades asociadas a un valor parcial, y un valor parcial es un subconjunto de elementos del dominio de un atributo, y sólo uno de estos elementos corresponde al valor del valor parcial. De ahí se parte para mostrar las definiciones siguientes mostradas en [1].

**Probabilidad parcial Referencial (RPP)**
Es un vector de probabilidades entre la llave foránea que en este caso es $R\_i.K$ y la llave primaria a la que hace referencia ésta, en este caso $R\_j.K$, cada valor en el vector indica la probabilidad de que un valor indefinido o invalido dentro de $R\_i.K$ este asociado con un valor valido de $R\_j.K$
.

$$\sum_{k \in R_j[K]} p(k) = 1 \tag{2.2}$$

Cada llave referenciada puede tener un conjunto de RPP que satisfaga la ecuación anterior. Aunque la forma de asignar las probabilidades cuando se propusieron las Funciones de Agregación Extendida fue siguiendo la siguiente lógica, una alta probabilidad corresponde a una alta frecuencia en la llave foránea y una baja probabilidad a una baja frecuencia.

**Frecuencia ponderada RPP**
Teniendo un valor $k$ dentro de los valores de $R_j[K]$ y el número de registros con una referencia válida dentro de $R_i[K]$ se representa como $\eta = |\{r_i \in R_i | r_i[K] \in R_j[K]\}|$, entonces se define *p(k)* como:

$$p(k) = \begin{cases} \frac{|\sigma_{K=k}(R_i)|}{\eta} & si\ \ \eta \neq 0 \\ \\ \frac{1}{|R_j[K]|} & de\ \ otro\ \ modo \end{cases} \tag{2.3}$$

**RPP Completa**
Teniendo un valor $k$ dentro de los valores de $R_j[K]$ en esta RPP, no se satisfacen todas las reglas de integridad de la tabla, se asigna a cada valor invalido o indefinido la probabilidad referencial 1 como si fuese un valor válido, entonces se define *p(k)* como:

$$p(k) = 1 \tag{2.4}$$

**RPP Restringida**

Teniendo un valor $k$ dentro de los valores de $R_j[K]$ en esta RPP que tampoco satisface la integridad, se asigna a cada valor invalido o indefinido la probabilidad referencial a 0 como si fuese un valor invalido, entonces se define *p(k)* como:

$$p(k) = 0 \tag{2.5}$$

**Referencialidad RPP**

Teniendo un subconjunto $r_i[K]$ dentro de los valores de $R_i$, y un valor $k$ dentro de los valores de $R_j[K]$, se puede definir la referencialidad de un valor de llave foránea $r_i[K]$ respecto a $k$, $REF(r_i[K], k)$ como:

$$p(k) = \begin{cases} 1 & si \;\; r_i[K] = k \\ 0 & si \;\; r_i[K] \neq k \;\; y \;\; r_i[K] \in R_j[K] \\ p(k) & si \;\; r_i[K] \notin R_j[K] \end{cases} \tag{2.6}$$

Donde la probabilidad *p(k)* corresponde a un RPP dado.

## 2.3 Funciones de Agregación Extendida.

Las Funciones de Agregación Extendida se definen en los artículos [10] y [11], y en el artículo [1] se extendieron y se generalizaron. Tenemos entonces, que si existen dos tablas $R_i$ y $R_j$ con un campo $[K]$ que es llave primaria en $R_j$ y llave foránea en $R_i$, y una restricción de integridad entre estas dos tablas de la forma $R_i(K) \rightarrow R_j(K)$ donde esta restricción supongamos es violada o sea que la tabla $R_i$ tenga errores de integridad referencial, entonces se tiene un subconjunto de registros $r_i[K]$ perteneciente a $R_i[K]$ y un valor $k$ que pertenece al conjunto de valores dentro de $R_j[K]$, se definen las Funciones de Agregación Extendida bajo las RPP mostradas anteriormente de la siguiente forma:

**Conteo por llave primaria**

La siguiente expresión significa, el número de registros de la tabla $R_i$ que tienen como valor de su llave foranea $K$ el valor $k$, que equivale al conteo de registros dado que la llave primaria no es nula.

$$x\_count(R_i.PK, r_i[K] = k) = \sum_{r_i \in R_i} REF(r_i[K], k) \tag{2.7}$$

**Conteo por columna específica**

La siguiente expresión significa, el conteo de los registros de la columna $A$, dentro de la tabla $R_i$, esto respecto al valor $k$ de llave foránea $K$. Como lo marca la definición del estándar SQL [2] los registros con un valor invalido en la columna $A$ son ignoradas.

$$x\_count(R_i.A, r_i[K] = k) = \sum_{r_i \in R_i} REF(r_i[K], k) \tag{2.8}$$

**Suma por columna específica**

La siguiente expresión significa, la suma de los registros de la columna $A$, dentro de la tabla $R_i$, esto respecto al valor $k$ de llave foránea $K$. Como lo marca la definición del estándar SQL [2] los registros con un valor invalido en la columna $A$ son ignoradas.

$$x\_sum(R_i.A, r_i[K] = k) = \sum_{r_i \in R_i} r_i[A] * REF(r_i[K], k) \tag{2.9}$$

**Promedio por columna específica**
La siguiente expresión significa, el promedio de los registros de la columna $A$, dentro de la tabla $R_i$, esto respecto al valor $k$ de llave foránea $K$. En el caso especial del promedio puede ser calculado con sus contrapartes $x\_sum$ y $x\_count$, como dice el estándar SQL [2] se calcula el promedio de los valores de $A$ no nulos. Se tiene que tomar en cuenta que el número de registros que califican para avg() no es el mismo que registros que califican para count($K$), tomando en cuenta que $K$ es la llave foránea, cada referencia válida en $K$ asociada a un valor invalido en $A$ añade un 1 a count($K$) pero no al número de registros para avg($A$)

$$x\_avg(R_i.A, r_i[K] = k) = \begin{cases} \frac{x\_sum(R_i.A, r_i[K]=k)}{x\_count(R_i.A, r_i[K]=k)} & si\ \ x\_count(R_i.A, r_i[K] = k) no\ \ es\ \ \eta \\ \\ \eta \qquad de\ \ otro\ \ modo \end{cases} \tag{2.10}$$

**Valor máximo por columna específica**
La siguiente expresión se utiliza para encontrar el valor máximo de los registros de la columna $A$, dentro de la tabla $R_i$, esto respecto al valor $k$ de llave foránea $K$. La Función de Agregación Extendida para calcular el valor máximo utilizando la RPP Completa o la RPP Restringida es la siguiente:

$$x\_max(R_i.A, r_i[K] = k) = MAX(\{r_i[A] * REF(r_i[K], k) | r_i \in R_i\}) \tag{2.11}$$

Para calcular la Frecuencia ponderada RPP y la Frecuencia ponderada referencial se tiene que calcular la media del valor máximo $r_i[A]$ en $R_i$ de cada uno de los valores de la referencia $k$ que puede estar presente en el conjunto de registros con una referencia inválidad o indefinida en la llave foránea $R_i.[K]$, para esto se utilizan los siguientes conceptos:

$E_i$ es el conjunto de registros con una referencia inválida en la llave foránea $K$.

$$E_i = \{r_i | r_i \in R_i \wedge r_i[K] \notin R_j[K]\} \tag{2.12}$$

El número total de elementos con una referencia inválida en el campo de llave foránea $K$ es $e$.

$$e = |E_i| \tag{2.13}$$

El número de combinaciones de $m$ elementos elegidos entre los $e$ elementos que tienen un error de integridad inválido o indefinido.

$$\binom{e}{m} \tag{2.14}$$

Donde $m$ es el número de registros, respecto al RPP escogido que estarán presentes en el conjunto total de registros con errores de integridad referencial.

$$m = \lceil p(k) * e \rceil \tag{2.15}$$

La referencialidad de las referencias inválidas o indefinidas es usada aquí para determinar el promedio en que una referencia definida puede estar presente en $E_i$. Para determinar la forma en que los $r_i[A]$ valores en $E_i$ pueden estar presentes en esta media y obtener la media del máximo en $r_i[A]$ de todas las instancias de $R_i$ para un cierto valor $k$, se tiene que hacer lo siguiente:

Sea $(a_e)$ la secuencia de todos los valores $r_i[A]$ en $E_i$ y, en el caso de las funciones $w\_max()$, los elementos de $(a_e)$ ordenados en orden descendente. Si $\{r_i | r_i \in R_i \wedge r_i[K] = k\} \neq \emptyset$, en esta secuencia, los valores en $r_i[A]$ que cumplan la condición $r_i[A] \ < amax$, donde $amax = MAX(\{r_i[A] | r_i \in R_i \wedge r_i[K] = k\})$, entonces son substituidos por $amax$.

Entonces se tendría la fórmula:

$$w\_max(R_i.A, r_i[K] = k) = \begin{cases} MAX(\{r_i[A] | r_i \in R_i\}) & si \;\; e = 0 \\ \dfrac{\sum_{i=1}^{e-m+1} a_i * \binom{e-i}{m-1}}{\binom{e}{m}} \end{cases} \tag{2.16}$$

**Valor mínimo por columna específica**
La siguiente expresión se utiliza para encontrar el valor mínimo de los registros de la columna $A$, dentro de la tabla $R_i$, esto respecto al valor $k$ de llave foránea $K$. La Función de Agregación Extendida para calcular el valor mínimo utilizando la RPP Completa o la RPP Restringida es la siguiente:

$$x\_min(R_i.A, r_i[K] = k) = MIN(\{r_i[A] * REF(r_i[K], k) | r_i \in R_i\}) \tag{2.17}$$

Para calcular la Frecuencia ponderada RPP y la Frecuencia ponderada referencial se tendría que hacer lo mismo que en la función para el cálculo del máximo sólo que los valores de $(a_e)$ se ordenarían ascendentemente, y se aplicará la condición $r_i[A] > amin$, donde $amin = MIN(\{r_i[A] | r_i \in R_i \wedge r_i[K] = k\})$, cambiando por $amin$ si se cumple la condición.

Entonces se tendría la fórmula:

$$w\_min(R_i.A, r_i[K] = k) = \begin{cases} MIN(\{r_i[A] | r_i \in R_i\}) & si \;\; e = 0 \\ \dfrac{\sum_{i=1}^{e-m+1} a_i * \binom{e-i}{m-1}}{\binom{e}{m}} \end{cases} \tag{2.18}$$

## 2.4 Modificación a la función min() y max()

Debido a que se están manejando cantidades de datos muy grandes (2,999,303 registros con errores en la tabla con 5% de error de integridad referencial y 17,995,816 de registros con errores en la tabla con un 30% de error de integridad referencial) los diferentes sistemas que se utilizaron para el estudio tienen una limitante en el tamaño de sus variables y los cálculos que se pueden hacer sobre las mismas.

Como por ejemplo R que al calcular el factorial, sólo acepta valores menores a 170 y las Funciones de Agregación Extendida max() y min(), necesitan el cálculo del factorial del número total de errores en la tabla, siendo imposible para el sistema tal cálculo, esto es una limitante para estas Funciones de Agregación Extendida, ya que los cálculos que se llevan a cabo involucran tablas que sobrepasan los millones de datos con errores de integridad referencial, o también se tiene el caso de PostgreSQL en este sistema el valor de una variable no puede superar 1e+307 dígitos siendo también limitante para este estudio con las fórmulas max() y min() tal como se presentaron.

Con esto en mente se optó por la modificación de las Funciones de Agregación Extendida min() y max(), de manera que se le permitiera a los diferentes sistemas poder hacer los cálculos necesarios sobre cada una, el proceso de modificación fue el siguiente.

La fórmula original de la Función de Agregación Extendida para encontrar el valor máximo con respecto a la frecuencia ponderada escogida es la presentada en 2.19.

Tomando en cuenta sólo la parte del dividendo de la fórmula, con un 5% de errores, esto nos daría una variable $e = 2{,}999{,}303$ que representa a los registros con un error de integridad referencial y una $m$ con un valor de 50, entonces la notación combinatoria quedaría de la siguiente fórmula.

$$\binom{e}{m} = \frac{e!}{m!(e-m)!} = \frac{2999303!}{50!(2999303-50)!} = \frac{2999303!}{50!2999253!}$$

Como se puede observar esos cálculos no son posibles en los sistemas aquí presentados por lo que se optó por simplificar la función para obtener un cálculo que se pueda llevar acabo en los sistemas, entonces teniendo la función.

$$w\_max(R_i.A, r_i[K] = k) = \begin{cases} MAX(\{r_i[A] | r_i \in R_i\}) & si \;\; e = 0 \\ \dfrac{\sum_{i=1}^{e-m+1} a_i * \binom{e-i}{m-1}}{\binom{e}{m}} \end{cases} \tag{2.19}$$

Debemos tomar en cuenta que $m$ depende de $e$ y *p(k)* entonces en esta implementación siempre será menor a $e$ (en implementaciones más pequeñas podría darse el caso que $m<=e$), lo primero es poner la fórmula completa transformando a cálculos matemáticos las expresiones combinatorias.

$$\text{fw\_}max(R_i.A, r_i[K] = k) = \left\{ \frac{\sum_{i=1}^{e-m+1} a_i * \left( \frac{(e-i)!}{(m-1)!((e-i)-(m-1))!} \right)}{\frac{(e)!}{(m)!(e-m)!}} \right.$$

Tomando en cuenta que, dado que $e >= m$ y

$(e-i) = ((e-i)-(m-1))!((e-i)-(m-1)+1) * ((e-i)-(m-1)+2) * ((e-i)-(m-1)+3) * ... * ((e-i)-1) * (e-i)$

entonces

$$\text{fw\_}max(R_i.A, r_i[K] = k) =$$

$$\left\{ \frac{\sum_{i=1}^{e-m+1} a_i * \left( \frac{((e-i)-(m-1))!}{((e-i)-(m-1))!} \frac{(((e-i)-(m-1))+1) * \cdots * ((e-i)-1) * (e-i)}{(m-1)!} \right)}{\frac{(e-m)!}{(e-m)!} \frac{((e-m)+1)*((e-m)+2)*\cdots*(e-1)*(e)}{(m)!}} \right.$$

Por lo que tendríamos para el cálculo del máximo la siguiente expresión.

$$fw\_max(R_i.A, r_i[K] = k) = \left\{ \frac{\sum_{i=1}^{e-m+1} a_i * \left( \frac{(((e-i)-(m-1))+1) * \cdots * ((e-i)-1) * (e-i)}{(m-1)!} \right)}{\frac{((e-m)+1)*((e-m)+2)*\cdots*(e-1)*(e)}{(m)!}} \right. \tag{2.20}$$

en la cual se reducen los cálculos considerablemente.

Para el cálculo de min() se realizarán los mismos pasos, para dejar la fórmula básicamente de la misma forma que se hizo anteriormente con max() sólo que para el cálculo del min() se ordenan los valores con errores de integridad referencial en forma ascendente en $a_e$ y la comparación es con el valor mínimo respecto a cada $K$, pero la función en general quedara de la misma forma que la de max().

$$fw\_min(R_i.A, r_i[K] = k) = \left\{ \frac{\sum_{i=1}^{e-m+1} a_i * \left( \frac{(((e-i)-(m-1))+1) * \cdots * ((e-i)-1) * (e-i)}{(m-1)!} \right)}{\frac{((e-m)+1)*((e-m)+2)*\cdots*(e-1)*(e)}{(m)!}} \right. \tag{2.21}$$

# CAPÍTULO 3

# Entorno de experimentación

En este capítulo se describirá cada uno de los entornos de experimentación dentro de cada sistema. Además de la configuración de la computadora donde se desarrollaron los experimentos, la configuración de la base de datos, las tablas auxiliares que se utilizaron y la forma de implementación en cada sistema. Cada entorno y configuración individual de cada sistema será explicado a detalle en cada sección, los 4 sistemas se instalaron en la misma computadora en el mismo sistema operativo sin cambiar ninguna característica fuera de la configuración de cada sistema.

## 3.1 Plataforma utilizada

Para el cálculo de las Funciones de Agregación Extendida se utilizó una computadora de escritorio con las siguientes características: 16 Gb de RAM (4 memorias de 4Gb cada una) a una velocidad de 1600 MHz, un procesador AMD de 6 núcleos a una velocidad de 2.8 GHz cada núcleo, el sistema operativo elegido para manejar todos los sistemas fue Ubuntu 14.04.

Se utilizó una base de datos sintética TPCH con un factor de escala de 10, la cual tiene esquema mostrado en la figura 3.1.

Al esquema de la figura 3.1 se le agregaron 5 copias de la tabla *LINEITEM*, con el mismo número total de registros, pero cierta cantidad de estos tienen errores de integridad referencial que van desde el 5% hasta el 30%, el esquema de las tablas se muestra en la figura 3.2.

Para la aplicación de las Funciones de Agregación Extendida, se utilizarán las tablas *SUPPLIER* con su llave primaria *S_SUPPKEY* y la tabla *LINEITEM* con su llave foránea *L_SUPPKEY* haciendo referencia a la llave primaria de la tabla *SUPPLIER* y los cálculos se realizarán sobre la columna *L_EXTENDEDPRICE*, sobre la tabla *LINEITEM* se aplicarán diversos porcentajes de errores de integridad

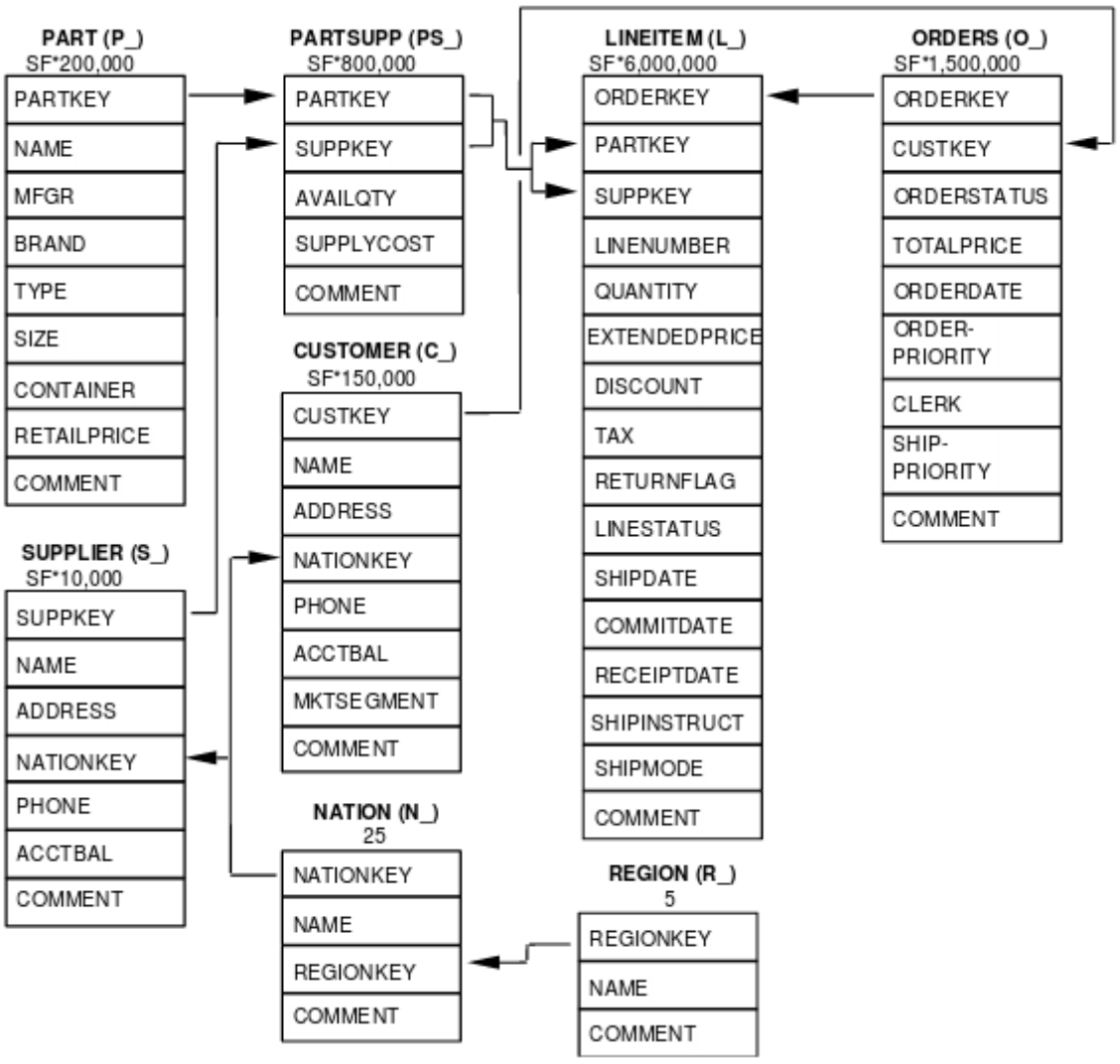


Figure 3.1: Esquema de la base de datos TPCH.

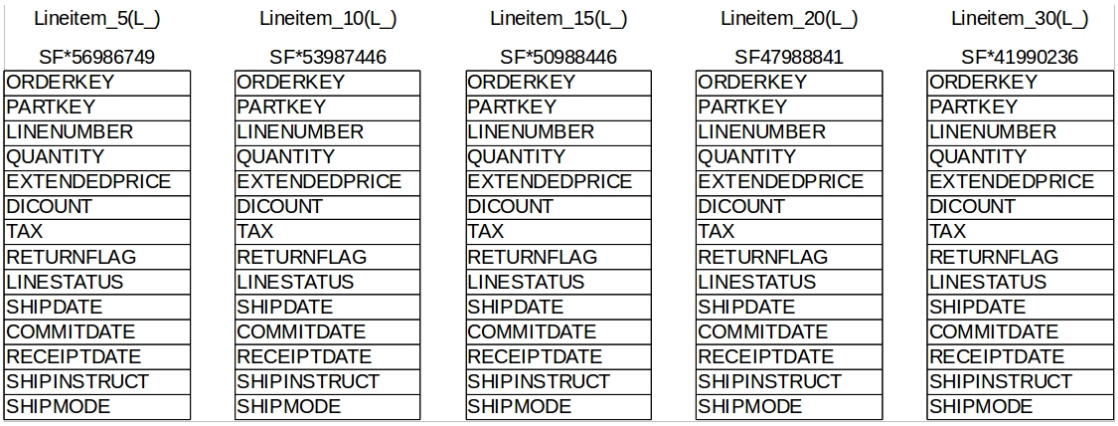


Figure 3.2: Esquema de las tablas con porcentajes de errores de integridad referencial.

referencial representados en las tablas que se mostraron en la figura 3.2, quedando de la siguiente manera.

Table 3.1: Porcentajes de error en tablas

| Porcentaje de error | Registros sin errores de integridad referencial | Registros con errores de integridad referencial |
|---|---|---|
| 5% | 56,986,749 | 2,999,303 |
| 10% | 53,987,446 | 5,998,606 |
| 15% | 50,988,144 | 8,997,908 |
| 20% | 47,988,841 | 11,997,211 |
| 30% | 41,990,236 | 17,995,816 |

## 3.2 Entorno SciDB

Para el desarrollo de este estudio se utilizó la versión 15.7 Community Edition de SciDB instalado como un servidor único, importando los archivos con extensión .tbl que proporciona TPCH para representar cada una de las tablas, para después insertarlos en un esquema de tipo arreglo, en este caso no hay reglas de integridad referencial que se puedan implementar directamente sobre los arreglos, así que cada archivo .tbl será un arreglo.

Debido a que SciDB no contiene funciones para el cálculo de factoriales, ciclos, ni tampoco algunas ecuaciones matemáticas necesarias en la función para el cálculo del valor máximo y mínimo, se optó por la instalación de R en su versión 3.4.0 y posteriormente dentro de R se instaló la librería correspondiente a SciDB que es SciDBR, esta librería tiene la opción de trabajar dentro de R los arreglos de SciDB. Dentro de las configuraciones especiales que se tienen que hacer dentro de SciDB están: la configuración del máximo número de impresiones ya que por defecto SciDB imprime sólo unos pocos resultados.

Para poder establecer una conexión entre SciDB y R dentro de un servidor único se tiene que instalar una herramienta extra, que es Shim, ésta es compatible con la versión 15.7 de SciDB, una vez instalado Shim sólo se tiene que poner en marcha, en este programa no hay necesidad de alguna configuración extra.

## 3.3 Entorno PostgreSQL

En el caso de PostgreSQL se utilizó la versión 9.3, se creó la base de datos y el esquema que utilizará la misma, se crearon las reglas de integridad referencial sobre las tablas que así lo ameritaron, también se importaron los archivos con extensión .tbl que proporciona TPCH para poder llenar las tablas. La única configuración especial que se tuvo que hacer dentro de PostgreSQL es la desactivación de la opción de paginación en los resultados.

Las ecuaciones matemáticas necesarias para el cálculo de las Funciones de Agregación Extendida max(), min() en este SGBD, se pueden realizar sin ningún problema. Por esta razón, no se tuvo que instalar ninguna herramienta extra, todos los cálculos necesarios estarán contenidos dentro de una función, que será llamada cuando sea necesario.

## 3.4 Entorno HP Vertica

Para HP Vertica se utilizó la versión 8.0.0-3, se creó la base de datos con ayuda de la herramienta administrativa proporcionada por el mismo sistema, el esquema y las reglas de integridad referencial que se utilizaron se programaron directo en la línea de comandos, se importaron archivos CSV creados con anterioridad en PostgreSQL, estos archivos representarán los datos que debe tener cada una de las tablas.

Las ecuaciones matemáticas necesarias para el cálculo del valor máximo y mínimo de las Funciones de Agregación Extendida no las soporta HP Vertica, así que se optó por una de las alternativas que recomienda HP Vertica para ampliar sus capacidades, en este caso la más recomendada es Python. Se utilizó la versión 2.7.6 de Python que viene instalada por defecto dentro del sistema operativo Ubuntu 14.04, se instaló el ODBC de Python para poder trabajar dentro de Python con las tablas de HP Vertica, además se configuraron los archivos de conexión, una vez configurado todo sólo se tuvo que establecer la conexión entre Python y HP Vertica desde la linea de comandos.

## 3.5 Entorno Apache Spark

Para Apache Spark se utilizó la versión 2.0.1, además del jdbc de PostgreSQL en su versión 9.4.1211 haciendo configuraciones extras en la memoria de ejecución, el número de ejecutores, la configuración del espacio del recolector de basura de java, la memoria del controlador, el número máximo de resultados, el tamaño máximo de estos resultados y el número máximo de núcleos del procesador a utilizar.

Se importaron los datos desde PostgreSQL utilizando el jdbc, después se guardaron estos en DataFrames, para su posterior almacenado dentro de Apache Spark en archivos con extensión .parquet. El proceso para acceder a los datos es leer los archivos con extensión . parquet, guardar el resultado de la lectura en DataFrames y después se crear vistas temporales para poder acceder a ellos por medio de consultas SQL.

Spark soporta las ecuaciones matemáticas necesarias para el cálculo de las Funciones de Agregación Extendida, sólo se tuvo que agregar una función extra para el cálculo del factorial, una vez creada ésta función, todos los cálculos matemáticos necesarios estarán contenidos dentro de funciones de Spark.

# CAPÍTULO 4

# Experimentos

En este capítulo se mostrará la implementación de las Funciones de Agregación Extendida en cada uno de los sistemas, con el fin de poder observar el comportamiento de cada sistema y conocer, cuál de éstos tiene un desempeño superior en cuestión de las características analizadas: tiempo de ejecución, consumo de memoria RAM, y porcentaje de procesador utilizado. Una vez que se tengan los resultados individuales de cada sistema se hará una comparativa conjunta en cada una de las secciones analizadas, con esto se pondrá en manifiesto cuál de los sistemas es el más adecuado para la ejecución de las Funciones de Agregación Extendida.

El motivo por el cual se presenta en este capítulo la implementación de las distintas Funciones de Agregación Extendida expresadas en consultas, es para efecto de demostrar que son comparables los resultados, dado que los datos utilizados para cada consulta son los mismos y las consultas se trataron de expresar de la misma forma en los diferentes sistemas utilizados.

Dado que las Funciones de Agregación Extendida internamente se compone de por lo menos estas 2 partes. Un conjunto consultas anidadas para el calculo de los valores con una referencia valida. Otro conjunto de consultas anidadas para el calculo de los valores con una referencia invalida. Se opto por convertir estos dos conjuntos cada uno en una tabla, los datos de esta separación se reportan en el conjunto de 4 tablas auxiliares que se crean al inicio de los experimentos en cada uno de los sistemas, Y los calculos para obtener el resultado final se realizan despues de crear estas tablas auxiliares.

## 4.1 Experimentos en SciDB

Se crearán arreglos adicionales, en uno de éstos estarán todos los registros con errores de integridad referencial de cada tabla *LINEITEM* ordenados de mayor a menor. En otro arreglo con la misma estructura del anterior sólo que los datos estarán ordenados de menor a mayor. En otro arreglo tendrá las filas que tengan una llave foránea válida de igual forma esto respecto a cada tabla *LINEITEM*. El último arreglo tendrá agrupados los valores por la llave foránea *L_SUPPKEY* respecto a cada tabla *LINEITEM*, con estos arreglos se podrán calcular con mayor rapidez y de forma óptima todas las Funciones de Agregación Extendida.

Las tablas de resultados contienen las siguientes columnas:

1. % de error: es el porcentaje de registros que tienen errores de integridad referencial en la tabla referenciante
2. Tiempo de ejecución: es el tiempo que tarda la consulta en ejecutarse
3. Memoria usada:Es cantidad de memoria RAM utilizada por la aplicación.
4. % de procesador: en el caso de SciDB utiliza 4 procesos para ejecutarse estos son SciDB-000-0-myd SciDB-000-1-myd SciDB-000-2-myd SciDB-000-3-myd

**Arreglo LI_SUPPLIER**

El arreglo *LI_SUPPLIER* contendrá las registros de cada uno de los 5 arreglos LINEITEM que tengan una referencia válida respecto al arreglo *SUPPLIER*, sólo se tomará de esta reunión los campos *L_SUPPKEY* de tipo entero y *L_EXTENDEDPRICE* de tipo flotante además de las dimensiones *n* e *ind* donde *ind* tendrá los mismos valores que el campo *L_SUPPKEY*, debido a que no hay una reunión natural en SciDB se optó por la búsqueda por índices.

store(redimension (apply (project (sort (filter (index_lookup (LINEITEM, project (SUPPLIER, S_SUPPKEY), LINEITEM.L_SUPPKEY, ind), ind¿=0), ind desc), LINEITEM.L_SUPPKEY, L_EXTENDEDPRICE), ind, LINEITEM.L_SUPPKEY), ¡L_SUPPKEY: int64, L_EXTENDEDPRICE : float¿ [n=0:*, 1000000, 0, ind=0:*, 1000000, 0]), LI_SUPPLIER_5);

Sólo se tendría que cambiar el nombre LINEITEM al nombre de la tabla que se va a analizar, los resultados de la consulta anterior aplicada a cada una de las 5 tablas se muestran en la siguiente tabla:

**Arreglo tm_ae_max**

Table 4.1: Arreglo LI_SUPPLIER

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | Usada GB | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 5923.346 | 15.50 | 150.6 | 149.6 | 152.6 | 145.9 |
| 10 | 5648.347 | 15.50 | 150.3 | 147.6 | 152.6 | 148.9 |
| 15 | 5484.515 | 15.50 | 151.6 | 153.9 | 142.0 | 152.3 |
| 20 | 5187.911 | 15.49 | 153.3 | 151.3 | 149.9 | 144.9 |
| 30 | 4850.912 | 15.44 | 154.9 | 150.6 | 142.0 | 150.9 |

El arreglo *tm_ae_max* contendrá las registros de los arreglos *LINEITEM* (5, 10, 15, 20, 30) que tengan un error de integridad referencial respecto al arreglo *SUPPLIER*, únicamente se tomará de esta reunión, el campo *L_EXTENDEDPRICE* de tipo flotante, y se ordenará este valor en forma descendente. Además se creará la variable $s$ con respecto a la dimensión $n$ que se crea automáticamente, que comenzará en 1 y terminará en el número de registros totales con un error de integridad referencial, igual que en el arreglo anterior, como no se cuenta con una reunión natural en SciDB se optó por una búsqueda por índices. Después de encontrar todos los índices, a aquellos que tengan un valor nulo(registros con errores de integridad referencial), se sustituirá su valor por uno muy grande tal que no se utilice en ninguna llave foránea *S_SUPPKEY*.

store( apply( sort( project( filter( sort( substitute( index_lookup( LINEITEM_5, project(SUPPLIER,S_SUPPKEY), LINEITEM_5.L_SUPPKEY, ind ), build( ¡val:int64¿[i=0:0,1,0],99999999999999999),ind ), ind desc ), ind=99999999999999999 ), L_EXTENDEDPRICE ), L_EXTENDEDPRICE desc ), s,n*1+1 ) ,tm_ae_max_5);

Al igual que en la consulta anterior, se tendrá que cambiar el nombre *LINEITEM* al nombre de la tabla que se pretenda analizar, los resultados de la consulta anterior aplicada a cada una de las 5 tablas se muestran en la siguiente tabla:

**Arreglo tm_ae_min**

Table 4.2: Arreglo tm_ae_max

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 3392.469 | 15.50 | 165.2 | 149.3 | 139.3 | 144.3 |
| 10 | 3492.073 | 15.51 | 141.0 | 161.6 | 140.0 | 151.3 |
| 15 | 3571.033 | 15.49 | 128.3 | 136.3 | 157.2 | 177.2 |
| 20 | 3687.316 | 15.52 | 152.9 | 149.9 | 142.0 | 151.6 |
| 30 | 3716.108 | 15.51 | 147.6 | 152.6 | 148.3 | 149.9 |

En el caso del arreglo *tm_ae_min* tendrá la misma estructura y los mismos campos que el arreglo *tm_ae_max* sólo que el campo *L_EXTENDEDPRICE* se ordenará en forma ascendente, ya que este arreglo se utilizará para el cálculo del valor mínimo, así que se utilizará como base el arreglo *tm_ae_max* y se ordenará con respecto a *L_EXTENDEDPRICE* y se insertará una nueva un nuevo valor $s$ con respecto a la dimensión $n$ la cual se crea automáticamente.

store(apply(sort(project(tm_ae_max,L_EXTENDEDPRICE),L_EXTENDEDPRICE asc), s,n*1+1 ),tm_ae_min);

Para el cálculo de cada porcentaje de error se tomará como base los arreglos de tm_ae_max, entonces los resultados son los siguientes:

Table 4.3: Arreglo tm_ae_min

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 34.157 | 2.98 | 100.1 | 100.1 | 100.1 | - - - - |
| 10 | 73.382 | 3.27 | 197.8 | 199.4 | 99.1 | 99.7 |
| 15 | 127.455 | 3.49 | 194.5 | 129.7 | 127.3 | 146.3 |
| 20 | 126.656 | 3.70 | 146.3 | 147.6 | 156.3 | 145.6 |
| 30 | 210.297 | 4.13 | 145.9 | 148.9 | 153.3 | 149.6 |

**Arreglo tm_fw**

En este arreglo se guardarán todos los datos necesarios para el cálculo de las funciones max() y min(),el primer paso es agrupar las registros del arreglo *LI_SUPPLIER* respecto a los valores de llave foránea *L_EXTENDEDPRICE* validos, una vez agrupado se calcula el valor máximo de cada agrupación y éste se guarda en la variable *maxxx*, de igual forma el valor mínimo se guarda en la variable *minnn*, se calcula el valor correspondiente a *p(k)*, se calcula el valor correspondiente a *m* y el valor correspondiente a *e*.

store( project( apply( cross_join( aggregate( LI_SUPPLIER, max(L_EXTENDEDPRICE) as maxxx, min(L_EXTENDEDPRICE) as minnn, count(L_EXTENDEDPRICE) as g_1, ind), cross_join( aggregate( LI_SUPPLIER, count(L_EXTENDEDPRICE) as t1),
aggregate( tm_ae_max, count(L_EXTENDEDPRICE) as e))) ,pk,g_1*1.0/t1, m, ceil((g_1*1.0/t1)*e), L_SUPPKEY, ind), L_SUPPKEY, pk, m, e, maxxx,minnn), tm_fw);

Los datos obtenidos al crear los arreglos respecto a cada porcentaje de error de integridad referencial son los siguientes:

Table 4.4: Arreglo tm_fw

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 43.564 | 4.42 | 91.1 | 90.4 | 90.7 | 91.4 |
| 10 | 40.987 | 4.38 | 80.0 | 84.0 | 80.3 | 85.0 |
| 15 | 38.892 | 4.31 | 89.5 | 78.3 | 77.6 | 84.2 |
| 20 | 36.466 | 4.16 | 87.2 | 86.5 | 90.5 | 90.2 |
| 30 | 33.69 | 4.02 | 86.2 | 77.3 | 76.6 | 77.9 |

En esta primera parte se tendrían los arreglos auxiliares, que separan los datos con errores de integridad referencial de los datos que no tienen errores de integridad referencial. En las siguientes implementaciones de las Funciones de Agregación Extendida se tendrá que tomar en cuenta que las tablas que se utilizan para estos cálculos ya están creadas, y se registraron los datos de las mismas.

### 4.1.1 Conteo por llave primaria

Para realizar el conteo en las Funciones de Agregación Extendida, respecto a la frecuencia ponderada, se utilizarán los arreglos *LI_SUPPLIER* con respecto al campo que representa la llave *L_SUPPKEY* y *tm_ae_max*, la consulta queda de la siguiente manera:

project(apply(cross_join( aggregate( LI_SUPPLIER, count(L_SUPPKEY), ind ), cross_join( project( apply( aggregate( tm_ae_max, count(*)), contador, count), contador), project( apply( aggregate(LI_SUPPLIER,count(*)), total, count), total))), fw_count, L_SUPPKEY_count + (contador*1.0*(L_SUPPKEY_count*1.0/total))), fw_count);

Se aplicará a cada uno de los porcentajes de error quedando los resultados de la siguiente manera:

Table 4.5: Conteo por llave primaria

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 28.958 | 4.33 | 89.5 | 88.5 | 85.2 | 92.2 |
| 10 | 27.163 | 4.24 | 84.0 | 91.6 | 88.3 | 91.0 |
| 15 | 25.351 | 4.08 | 91.5 | 76.6 | 85.2 | 77.2 |
| 20 | 23.641 | 3.10 | 86.7 | 59.1 | 61.8 | 91.7 |
| 30 | 21.54 | 3.63 | 81.2 | 65.3 | 62.6 | 92.1 |

#### Conteo total por llave primaria

Para el conteo total, respecto a la llave primaria se utiliza como base la consulta mostrada con anterioridad, y se modifica; agregando una sumatoria de todos los valores anteriores, la consulta entonces queda de la siguiente forma:

aggregate(project(apply(cross_join( aggregate( LI_SUPPLIER, count(L_SUPPKEY), ind ), cross_join( project( apply( aggregate( tm_ae_max,count(*)), contador, count), contador), project( apply( aggregate( LI_SUPPLIER,count(*)), total, count), total))), fw_count, L_SUPPKEY_count + (contador * 1.0 * (L_SUPPKEY_count * 1.0 / total))), fw_count), sum(fw_count) as fw_count);

Tomando cada uno de los 5 porcentajes de error diferente, los resultados de la consulta son los mostrados en la siguiente tabla:

Table 4.6: Conteo total por llave primaria

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 29.908 | 4.32 | 91.9 | 82.6 | 89.5 | 84.6 |
| 10 | 27.926 | 4.29 | 87.5 | 88.5 | 82.5 | 89.8 |
| 15 | 26.162 | 4.19 | 83.1 | 91.4 | 91.7 | 83.1 |
| 20 | 24.471 | 4.17 | 87.1 | 92.1 | 82.8 | 86.8 |
| 30 | 22.279 | 4.01 | 86.3 | 82.0 | 75.4 | 77.4 |

### 4.1.2 Conteo por columna especificada

En este caso la columna por la que se va a calcular el conteo, respecto a la frecuencia ponderada es *L_EXTENDEDPRICE*, la consulta es muy parecida a la que se muestra en la sección anterior, solo que aquí no se utiliza la columna que representa la llave primaria, se utiliza otra de las disponibles en el arreglo *LI_SUPPLIER*, la consulta entonces queda de la siguiente forma:

project(apply(cross_join(aggregate( LI_SUPPLIER, count(L_EXTENDEDPRICE), ind ), cross_join( project( apply( aggregate( tm_ae_max, count(L_EXTENDEDPRICE)), contador, L_EXTENDEDPRICE_count), contador), project( apply( aggregate( LI_SUPPLIER, count(L_EXTENDEDPRICE)), total, L_EXTENDEDPRICE_count), total))), fw_count, L_EXTENDEDPRICE_count + (contador * 1.0 * (L_EXTENDEDPRICE_count * 1.0 /total))),fw_count);

La tabla de los resultados para cada uno de los 5 porcentajes de error queda de la siguiente forma:

Table 4.7: Conteo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 29.638 | 4.34 | 90.1 | 82.8 | 90.7 | 81.5 |
| 10 | 27.47 | 4.23 | 77.6 | 86.2 | 60.3 | 72.3 |
| 15 | 25.69 | 4.16 | 85.3 | 80.7 | 77.3 | 84.7 |
| 20 | 24.106 | 4.09 | 79.3 | 83.6 | 66.3 | 73.0 |
| 30 | 22.023 | 3.89 | 73.9 | 83.2 | 80.5 | 90.8 |

#### Conteo total por columna especificada

Al igual que la sección anterior, para el conteo total respecto al mismo arreglo y a la misma columna *L_EXTENDEDPRICE*, se utilizará como base la consulta mostrada arriba, a la cual se le agregará una sumatoria de todos los valores, la consulta quedaría de la siguiente manera:

aggregate(project(apply(cross_join(aggregate( LI_SUPPLIER, count(L_EXTENDEDPRICE), ind ), cross_join( project( apply( aggregate( tm_ae_max,count( L_EXTENDEDPRICE)), contador, L_EXTENDEDPRICE_count), contador), project( apply( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE)), total, L_EXTENDEDPRICE_count), total))), fw_count, L_EXTENDEDPRICE_count + (contador * 1.0 * (L_EXTENDEDPRICE_count * 1.0/total))), fw_count), sum(fw_count) as fw_count);

Aplicando la consulta anterior a cada uno de los 5 porcentajes de error, los resultados se muestran en la siguiente tabla:

Table 4.8: Conteo total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 30.149 | 4.39 | 80.6 | 86.5 | 83.5 | 73.6 |
| 10 | 28.573 | 4.33 | 87.1 | 79.8 | 71.2 | 70.2 |
| 15 | 26.441 | 4.30 | 66.3 | 70.9 | 89.8 | 90.5 |
| 20 | 24.789 | 4.16 | 76.0 | 79.7 | 80.0 | 71.4 |
| 30 | 22.755 | 3.93 | 73.9 | 72.9 | 80.5 | 87.1 |

### 4.1.3 Suma por columna especificada

Para el caso del cálculo de la suma, respecto a la frecuencia ponderada, se utilizará la columna *L_EXTENDEDPRICE* del arreglo *LI_SUPPLIER* y el arreglo *tm_ae_max* con su columna *L_EXTENDEDPRICE*, entonces la consulta queda de la siguiente forma:

project( apply( cross_join( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE), sum( L_EXTENDEDPRICE) as total_group, ind ), cross_join( project( apply( aggregate( tm_ae_max, sum( L_EXTENDEDPRICE)), suma, L_EXTENDEDPRICE_sum), suma), project( apply( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE)), total, L_EXTENDEDPRICE_count), total))), fw_sum, total_group + (suma * 1.0 * (L_EXTENDEDPRICE_count * 1.0 / total))), fw_sum);

La tabla de los resultados para cada uno de los 5 porcentajes de error queda de la siguiente forma:

Table 4.9: Suma por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 33.598 | 4.32 | 84.8 | 90.1 | 67.6 | 74.2 |
| 10 | 31.692 | 4.32 | 86.8 | 89.4 | 81.5 | 78.8 |
| 15 | 29.699 | 4.13 | 89.9 | 85.9 | 89.3 | 81.3 |
| 20 | 27.497 | 4.14 | 78.6 | 83.9 | 81.0 | 80.0 |
| 30 | 25.179 | 3.98 | 84.7 | 80.0 | 87.0 | 83.0 |

#### Suma total por columna especificada

Para el cálculo de la suma total se utiliza como base la consulta para la Suma por columna especificada usada anteriormente, solo que a ésta se le agregará una sumatoria de todos los valores que mostró, entonces, la consulta queda de la siguiente manera:

aggregate( project( apply( cross_join( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE), sum( L_EXTENDEDPRICE) as total_group, ind ), cross_join( project( apply( aggregate( tm_ae_max, sum( L_EXTENDEDPRICE)), suma, L_EXTENDEDPRICE_sum), suma), project( apply( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE)), total, L_EXTENDEDPRICE_count), total))), fw_sum, total_group + ( suma * 1.0 * ( L_EXTENDEDPRICE_count * 1.0 / total))), fw_sum), sum( fw_sum) as fw_sum);

Aplicando la consulta anterior a cada uno de los 5 porcentajes de error, los resultados se muestran en la siguiente tabla:

Table 4.10: Suma total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 35 | 4.47 | 85.5 | 85.8 | 89.5 | 84.8 |
| 10 | 32.779 | 4.26 | 82.9 | 81.6 | 86.2 | 87.9 |
| 15 | 30.868 | 4.27 | 74.6 | 87.9 | 75.3 | 89.6 |
| 20 | 28.75 | 4.17 | 80.2 | 81.2 | 84.2 | 88.5 |
| 30 | 26.492 | 4.12 | 81.3 | 91.0 | 91.0 | 90.6 |

### 4.1.4 Promedio por columna especificada

En el caso del cálculo del promedio, respecto a la frecuencia ponderada, se utilizarán las consultas vistas con anterioridad, tanto la Suma por columna especificada, como el Conteo por columna especificada, las dos aplicadas a la misma columna *L_EXTENDEDPRICE* en el arreglo *LI_SUPPLIER* y a la columna *L_EXTENDEDPRICE* en el arreglo *tm_ae_max*, una vez que tengan los resultados de las dos operaciones, se dividirá la suma entre el conteo, con esto se obtendrá el promedio, quedando la consulta de la siguiente forma:

project( apply( cross_join( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE) as t_g_c, sum( L_EXTENDEDPRICE) as t_g_s, ind ), cross_join( project( apply( aggregate( tm_ae_max, sum( L_EXTENDEDPRICE), count( L_EXTENDEDPRICE)), suma, L_EXTENDEDPRICE_sum, contador, L_EXTENDEDPRICE_count), suma, contador), project( apply( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE)), total, L_EXTENDEDPRICE_count), total))), fw_avg, ( t_g_s + (suma * 1.0 * (t_g_c * 1.0 / total ))) / ( t_g_c + ( contador * 1.0 * ( t_g_c * 1.0 / total)))), fw_avg);

La tabla de los resultados para cada uno de los 5 porcentajes de error queda de la siguiente forma:

Table 4.11: Promedio por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 33.621 | 4.39 | 100.1 | 100.1 | 100.1 | 99.8 |
| 10 | 31.609 | 4.28 | 86.8 | 89.8 | 89.4 | 89.1 |
| 15 | 29.537 | 4.20 | 83.7 | 83.7 | 89.0 | 82.3 |
| 20 | 27.565 | 4.10 | 88.6 | 89.6 | 84.6 | 86.3 |
| 30 | 25.385 | 4.04 | 90.6 | 91.3 | 83.7 | 90.3 |

**Promedio total por columna especificada**

Para el cálculo del promedio total primero se calculará el Conteo total por columna especificada junto con la Suma total por columna especificada una vez que se tienen esos dos resultados se dividirá la suma total entre el conteo total, y así se obtendrá el promedio total, la consulta queda de la siguiente manera:

project( apply( aggregate( apply( cross_join( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE) as t_g_c, sum( L_EXTENDEDPRICE) as t_g_s, ind ), cross_join( project( apply( aggregate( tm_ae_max, sum( L_EXTENDEDPRICE), count( L_EXTENDEDPRICE)), suma, L_EXTENDEDPRICE_sum, contador, L_EXTENDEDPRICE_count), suma, contador), project( apply( aggregate( LI_SUPPLIER, count( L_EXTENDEDPRICE)), total, L_EXTENDEDPRICE_count), total))), fw_sum, t_g_s + ( suma * 1.0 * (t_g_c * 1.0 / total)), fw_count, t_g_c + ( contador * 1.0 * (t_g_c * 1.0 / total))), sum( fw_sum) as fw_sum, sum( fw_count) as fw_count), fw_avg, fw_sum / fw_count), fw_avg);

Aplicando la consulta anterior a cada uno de los 5 porcentajes de error, los resultados se muestran en la siguiente tabla:

Table 4.12: Promedio total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | | | |
|---|---|---|---|---|---|---|
| | | | SciDB-000-0-myd | SciDB-000-1-myd | SciDB-000-2-myd | SciDB-000-3-myd |
| 5 | 35.58 | 4.35 | 82.2 | 87.2 | 90.8 | 90.5 |
| 10 | 33.867 | 4.31 | 88.8 | 84.2 | 82.9 | 85.2 |
| 15 | 31.608 | 4.19 | 86.7 | 77.4 | 80.4 | 79.7 |
| 20 | 29.616 | 4.15 | 90.6 | 91.3 | 81.3 | 88.6 |
| 30 | 27.531 | 4.02 | 90.5 | 90.5 | 85.5 | 87.2 |

### 4.1.5 Valor máximo por columna especificada

Para el cálculo de esta Función de Agregación Extendida, respecto a la frecuencia ponderada, se utilizarán otras variables que están contenidos en el arreglo *tm_fw* en el que los valores ya están agrupados respecto a la llave foránea *L_SUPPKEY* y los valores necesarios ya están calculados, debido a que esta función requiere muchos cálculos matemáticos que no soporta directamente SciDB se optó por utilizar la librería correspondiente en R, que es SciDBR para los cálculos, entonces la función queda interpretada de la siguiente manera:

```
pmt<-proc.time()
x<-scidb("scan(tm_fw)")
x1<-data.frame(x[])
y<-scidb("scan(tm_ae_max)")
y1<-data.frame(y[])
for(ind in 1:100){

    varx<-0
    varr<-1
    a<-0
    b<-0
    e<-x1[ind,7]
    m<-x1[ind,6]
    fac<-factorial(m-1)
    if((e-1)==((e-1)-(m-1))){

        for(n in 1:(e-m-1)){

            if(y1[n,2]<=x1[ind,8]){

                varx<-varx+(x1[ind,8]*((e-m-1)-n+1))
                n<-(e-m-1)
            }else{

                varx<-varx+y1[n,2]
            }
        }
    }else{

        a<-fac
        while(a<(1E+200) && b<(m-2)){

            a<-a*(e-b)
```

```
        b<-b+1
      }
      varr<-varr*(((e-1)-(m-1)+1)/a)
      a<-1
      for(i in ((e-1)-(m-1)+2):(e-1)){

        while(a<1E+1 && b<(m-2)&& varr>7E+290){

          a<-a*(e-b)
          b<-b+1
        }
        varr<-varr*i/a
        a<-1;
      }
      varx<-varx+(y1[1,2]*varr)
      for(n in 2:(e-m-1)){

        if(varr>=3.47822214672238E-319){

          varr<-varr/(e-n+1)*((e-n)-(m-1)+1)
        }
        if(y1[n,2]<=x1[ind,8]){

          varx<-varx+(x1[ind,8]*varr)
        }else{

          varx<-varx+(y1[n,2]*varr)
        }
      }
    }
    varr<-1
    varr<-varr*((e-m+1)/(factorial(m)))
    for(i in (e-m+2):(e-b)){

      varr<-varr*i
    }
    varx<-varx/varr
    print(paste("ind= ",ind,"maxxx= ",x1[ind,8]," fw=",varx))
}
proc.time()-pmt
```

En este caso debido a las limitaciones de los sistemas en los que se basó este análisis, el cálculo del valor máximo en la tabla de 30% de error utilizaba variables tan grandes que ninguno de los sistemas las pudo soportar, por eso se optó por hacer el análisis hasta el 20% de error de integridad referencial, y por las limitaciones de tiempo solo se analizarán los 100 primeros elementos del arreglo, entonces, la tabla de los resultados para cada uno de los 4 porcentajes de error queda de la siguiente forma:

Table 4.13: Valor máximo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | |
|---|---|---|---|---|
| | | | Cargar datos desde SciDB a R | Ejecucion en R |
| 5 | 16623.129 | 1.47 | 100.4 | 100.0 |
| 10 | 33251.89 | 1.43 | 100.7 | 100.0 |
| 15 | 49791.703 | 2.09 | 100.7 | 100.0 |
| 20 | 68192.586 | 1.35 | 100.1 | 100.0 |

### 4.1.6 Valor mínimo por columna especificada

Para el cálculo del valor mínimo de la Función de Agregación Extendida, respecto a la frecuencia ponderada, se utilizarán las variables que se calcularon en el arreglo *tm_fw* donde los valores ya se encuentran agrupados, de igual forma debido a que los cálculos matemáticos que se necesitan para encontrar el valor mínimo no los soporta directamente SciDB, aquí también se optó por usar la librería SciDBR, entonces la función queda interpretada de la siguiente manera:

```
pmt<-proc.time()
x<-scidb("scan(tm_fw)")
x1<-data.frame(x[])
y<-scidb("scan(tm_ae_min)")
y1<-data.frame(y[])
for(ind in 1:100){

    varx<-0
    varr<-1
    a<-0
    b<-0
    e<-x1[ind,7]
    m<-x1[ind,6]
    fac<-factorial(m-1)
    if((e-1)==((e-1)-(m-1))){

        for(n in 1:(e-m-1)){

            if(y1[n,2]>=x1[ind,9]){

                varx<-varx+(x1[ind,9]*((e-m-1)-n+1))
                n<-(e-m-1)
            }else{

                varx<-varx+y1[n,2]
            }
        }
    }else{

        a<-fac
        while(a<(1E+200) && b<(m-2)){

            a<-a*(e-b)
            b<-b+1
        }
```

```
        varr<-varr*(((e-1)-(m-1)+1)/a)
        a<-1
        for(i in ((e-1)-(m-1)+2):(e-1)){

            while(a<1E+1 && b<(m-2)&& varr>7E+290){

                a<-a*(e-b)
                b<-b+1
            }
            varr<-varr*i/a
            a<-1;
        }
        varx<-varx+(y1[1,2]*varr)
        for(n in 2:(e-m-1)){

            if(varr>=3.47822214672238E-319){

                varr<-varr/(e-n+1)*((e-n)-(m-1)+1)
            }
            if(y1[n,2]¿=x1[ind,9]){

                varx<-varx+(x1[ind,9]*varr)
            }else{

                varx<-varx+(y1[n,2]*varr)
            }
        }
    }
    varr<-1
    varr<-varr*((e-m+1)/(factorial(m)))
    for(i in (e-m+2):(e-b)){

        varr<-varr*i
    }
    varx<-varx/varr
    print(paste("ind= ",ind,"maxxx= ",x1[ind,9]," fw=",varx))
}
proc.time()-pmt
```

Igual que el caso anterior, para poder calcular el valor mínimo respecto al arreglo con un 30% de error de integridad referencial se tendrían que utilizar variables muy grandes que rebasarían el limite permitido en el sistema utilizado para realizar este análisis, entonces con esto en mente se optó por analizar solo los arreglos con un 5%, 10%, 15% y 20% de error de integridad referencial y solo hacer el cálculo a los 100 primeros elementos, entonces la tabla de los resultados para cada porcentajes de error queda de la siguiente forma:

Table 4.14: Valor mínimo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | |
|---|---|---|---|---|
| | | | Cargar datos desde SciDB a R | Ejecucion en R |
| 5 | 16607.616 | 1.83 | 100 | 100.0 |
| 10 | 33195.246 | 1.96 | 100 | 100.0 |
| 15 | 49824.390 | 2.30 | 100 | 100.0 |
| 20 | 66357.798 | 2.51 | 100 | 100.0 |

## 4.2 Experimentos en PostgreSQL

Además de las tablas que proporciona TPCH, se crearon tablas auxiliares extras, la primera tabla contendrá todos los registros con errores de integridad referencial ordenados de mayor a menor. La segunda tabla tendrá la misma estructura que la primera y los mismos datos la única diferencia es que estos datos estarán ordenados de menor a mayor. La tercer tabla tendrá todos los registros que tengan una llave foránea válida. La cuarta tabla tendrá los registros con referencias válidas respecto a la llave foránea, estos datos estarán agrupados por la llave *L_SUPPKEY*, además de que ésta contendrá los valores que ayudarán a calcular más rápido las Funciones de Agregación Extendida max() y min().

Las tablas de resultados contienen las siguientes columnas:

1. % de error: es el porcentaje de registros que tienen errores de integridad referencial en la tabla referenciante
2. Tiempo de ejecución: es el tiempo que tarda la consulta en ejecutarse
3. Memoria usada:Es cantidad de memoria RAM utilizada por la aplicación.
4. % de procesador: Es el porcentaje de procesador que utiliza la consulta al ejecutarse

**Tabla LI_SUPPLIER**

Como primer paso se crearán las tablas *LI_SUPPLIER* en las cuales se guardarán las registros de las tablas *LINEITEM* que tengan una referencia válida respecto a la tabla *SUPPLIER*, tomando solo los campos *L_SUPPKEY* de tipo entero y *L_EXTENDEDPRICE* de tipo flotante, esto se realizará mediante una reunión natural sobre los campos *L_SUPPKEY* de la tabla *LINEITEM* y *S_SUPPKEY* de la tabla *SUPPKEY*.

create table LI_SUPPLIER as select LINEITEM.L_SUPPKEY, L_EXTENDEDPRICE from SUPPLIER inner join LINEITEM on SUPPLIER.S_SUPPKEY = LINEITEM.L_SUPPKEY order by LINEITEM.L_SUPPKEY desc;

Únicamente se tendrá que cambiar el nombre de la tabla por el nombre de la tabla correspondiente a cada porcentaje de error a analizar, los resultados de la consulta anterior aplicado a cada una de las 5 tablas se muestran en la siguiente tabla:

Table 4.15: Tabla LI_SUPPLIER

| % de error | Tiempo de ejecución(Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 310.53 | 12.34 | 93.2 |
| 10 | 292.27 | 12.48 | 99.9 |
| 15 | 276.46 | 12.31 | 99.9 |
| 20 | 278.15 | 12.20 | 100.0 |
| 30 | 292.75 | 11.24 | 99.7 |

**Tabla tm_ae_max**

La tabla *tm_ae_max* tendrá las registros de las tablas *LINEITEM* que tengan una referencia inválida respecto a la tabla *SUPPLIER*, de esta reunión se tomarán los campos *L_EXTENDEDPRICE* de tipo flotante, y éste se ordenará en forma descendente, y se agregará una columna *S* que representará un conteo de las registros que haya en la consulta, esto se realizará mediante una reunión a la derecha sobre los campos *L_SUPPKEY* de la tabla *LINEITEM* y *S_SUPPKEY* de la tabla *SUPPKEY* donde se tomarán solo las registros de *L_SUPPKEY* que tengan una referencia nula (registros con errores de integridad referencial).

create table tm_ae_max as select L_EXTENDEDPRICE, row_number() over() as s from (select L_EXTENDEDPRICE from SUPPLIER right join LINEITEM on SUPPLIER.S_SUPPKEY=LINEITEM.L_SUPPKEY where LINEITEM.L_SUPPKEY is null order by L_EXTENDEDPRICE desc) as sub1;

Para crear la tabla correspondiente a cada uno de los porcentajes de error, se cambiará el nombre *LINEITEM* al nombre de la tabla que se pretende analizar, los resultados de la consulta anterior aplicada a cada una de las 5 tablas se muestran en la siguiente tabla:

Table 4.16: Tabla tm_ae_max

| % de error | Tiempo de ejecución(Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 162.18 | 9.20 | 83.1 |
| 10 | 112.89 | 9.46 | 50.9 |
| 15 | 127.25 | 9.77 | 99.7 |
| 20 | 157.08 | 10.14 | 99.7 |
| 30 | 216.7 | 10.67 | 99.2 |

**Tabla tm_ae_min**

La tabla *tm_ae_min* tiene la misma estructura y los mismos datos que la tabla anterior, la única diferencia es que el campo *L_EXTENDEDPRICE* se ordena en forma ascendente, ya que se utilizará para calcular el valor mínimo, así que se utilizará la tabla *tm_ae_max* como base, ordenando el campo *L_EXTENDEDPRICE* en forma ascendente, y se calculará un nuevo valor para *s* que empezará en 1 y terminara con el numero total de registros en la tabla.

create table tm_ae_min as select L_EXTENDEDPRICE, row_number() over() as s from (select l_extendedprice from tm_ae_max order by l_extendedprice asc)as sub1;

Para crear la tabla correspondiente a cada uno de los porcentajes de error, se cambiará el nombre *LINEITEM* al nombre de la tabla que se va a analizar, los resultados de la consulta anterior aplicada a cada una de las 5 tablas se muestran en la siguiente tabla:

Table 4.17: Tabla tm_ae_min

| % de error | Tiempo de ejecución(Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 07.94 | 1.53 | 100.0 |
| 10 | 16.50 | 1.90 | 99.9 |
| 15 | 25.39 | 2.25 | 100.0 |
| 20 | 39.09 | 2.69 | 100.0 |
| 30 | 60.68 | 3.35 | 100.0 |

**Tabla tm_fw**

Por último se tiene la tabla *tm_fw*, ésta tabla tendrá los datos necesarios para el cálculo de las funciones max() y min(), los datos estarán agrupados respecto a los valores de llave foránea *L_SUPPKEY* con una referencia hacia la llave foranea válida, se calcularán tanto el valor máximo que se guardará en la variable *maxxx*, así como también el valor mínimo que se guardará en la variable *minnn*, además de las distintas variables que se necesitan para el calculo de dichas funciones.

create table tm_fw as select L_SUPPKEY, count( l_extendedprice) * 1.0 / t1 as pk, ceil( ( count( l_extendedprice) * 1.0 / t1) * e) as m, e, max( l_extendedprice) as maxxx, min( l_extendedprice) as minnn from LI_SUPPLIER cross join ( select count( l_extendedprice) as t1 from LI_SUPPLIER) as sub1 cross join (select count( l_extendedprice) as e from *tm_ae_max*) as sub2 group by L_SUPPKEY, e, t1;

Para crear la tabla correspondiente a cada uno de los porcentajes de error, se cambiará el nombre *LI_SUPPLIER* y *tm_ae_max* al nombre de la tabla que se va a analizar, los resultados de la consulta anterior aplicada a cada una de las 5 tablas

se muestran en la siguiente tabla:

Table 4.18: Tabla tm_fw

| % de error | Tiempo de ejecución(Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 242.24 | 5.50 | 100.0 |
| 10 | 235.21 | 5.40 | 100.0 |
| 15 | 225.14 | 5.28 | 100.0 |
| 20 | 202.42 | 5.18 | 100.0 |
| 30 | 159.21 | 4.94 | 100.0 |

En esta primera parte se tendrían las tablas auxiliares, que separan los datos con errores de integridad referencial de los datos que no tienen errores de integridad referencial. En las siguientes implementaciones de las Funciones de Agregación Extendida se tendrá que tomar en cuenta que las tablas que se utilizan para estos cálculos ya están creadas, y se registraron los datos de las mismas.

### 4.2.1 Conteo por llave primaria

El conteo en las Funciones de Agregación Extendida, respecto a la frecuencia ponderada, utiliza las tablas *LI_SUPPLIER* respecto su columna *L_SUPPKEY* y la tabla *tm_ae_max*, la consulta entonces queda de la siguiente manera:

Select L_SUPPKEY, count( L_SUPPKEY) + ( contador * 1.0 * ( count( L_SUPPKEY) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count(*) as total from LI_SUPPLIER) as sub1 cross join (select count(*) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error se muestran en la siguiente tabla:

Table 4.19: Conteo por llave primaria

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 228.81 | 5.11 | 100.0 |
| 10 | 231.06 | 4.98 | 100.0 |
| 15 | 203.86 | 4.86 | 100.0 |
| 20 | 208.27 | 4.74 | 100.0 |
| 30 | 205.15 | 4.50 | 99.7 |

**Conteo total por llave primaria**

Para realizar el conteo total por llave primaria se utilizará la consulta anterior, a la que se le agregará una sumatoria de todos los resultados mostrados en la columna *fw_count*, la consulta entonces quedaría de la siguiente forma:

select sum( fw_count) as fw_count_total from ( select L_SUPPKEY, count( L_SUPPKEY) + ( contador * 1.0 * ( count( L_SUPPKEY) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count(*) as total from LI_SUPPLIER) as sub1 cross join ( select count(*) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total) as sub3;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.20: Conteo total por llave primaria

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 195.65 | 5.06 | 100.0 |
| 10 | 174.85 | 4.94 | 100.0 |
| 15 | 164.11 | 4.82 | 100.0 |
| 20 | 161.88 | 4.70 | 100.0 |
| 30 | 145.25 | 4.46 | 100.0 |

### 4.2.2 Conteo por columna especificada

Para el cálculo del conteo por columna especificada, la columna que se escogió para este conteo, respecto a la frecuencia ponderada es *L_EXTENDEDPRICE*, la consulta es muy parecida a la mostrada anteriormente sólo que en la anterior se utiliza la llave primaria, mientras que en este conteo se utiliza cualquier otra tabla disponible en *LI_SUPPLIER*, la consulta entonces queda de la siguiente manera:

select L_SUPPKEY, count( l_extendedprice) + ( contador * 1.0 * ( count( l_extendedprice) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select count( l_extendedprice) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.21: Conteo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 255.97 | 5.49 | 100.0 |
| 10 | 222.87 | 5.35 | 100.0 |
| 15 | 212.70 | 5.21 | 100.0 |
| 20 | 199.93 | 5.06 | 100.0 |
| 30 | 170.26 | 4.78 | 100.0 |

#### Conteo total por columna especificada

Para realizar el conteo total por columna especificada, que en este caso se realiza sobre la columna *L_EXTENDEDPRICE*, se tomará como base la consulta anterior y se le agregará una sumatoria de todos los datos mostrados en la columna *fw_count*, entonces la consulta queda de la siguiente manera:

select sum( fw_count) as fw_count_total from ( select L_SUPPKEY, count( l_extendedprice) + (contador * 1.0 * ( count( l_extendedprice) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select count( l_extendedprice) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total) as sub3;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.22: Conteo total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 235.48 | 5.52 | 100.0 |
| 10 | 211.03 | 5.37 | 100.0 |
| 15 | 204.39 | 5.23 | 100.0 |
| 20 | 183.88 | 5.09 | 100.0 |
| 30 | 148.88 | 4.80 | 100.0 |

### 4.2.3 Suma por columna especificada

Para el cálculo de la suma por columna especificada, respecto a la frecuencia ponderada, se utilizará como base la columna *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, la consulta entonces queda de la siguiente manera:

select L_SUPPKEY, sum( l_extendedprice) + coalesce( ( suma * 1.0 * ( count( l_extendedprice) * 1.0 / total)), 0) as fw_sum from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join (select sum( l_extendedprice) as suma from tm_ae_max) as sub2 group by L_SUPPKEY, suma, total;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.23: Suma por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 258.82 | 5.50 | 100.0 |
| 10 | 228.47 | 5.36 | 100.0 |
| 15 | 217.96 | 5.22 | 100.0 |
| 20 | 192.28 | 5.07 | 100.0 |
| 30 | 164.77 | 4.78 | 100.0 |

**Suma total por columna especificada**

Para el cálculo de la suma total por columna especificada, se utiliza la columna *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER*. Se toma como base la consulta anterior y se le agrega una sumatoria de todos los datos mostrados en la columna *fw_sum*, la consulta queda de la siguiente manera:

select sum( fw_sum) as fw_sum_total from( select L_SUPPKEY, sum( l_extendedprice) + coalesce( ( suma*1.0*( count( l_extendedprice) * 1.0/total)), 0) as fw_sum from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select sum( l_extendedprice) as suma from tm_ae_max) as sub2 group by L_SUPPKEY, suma, total) as sub3;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.24: Suma total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 265.12 | 5.28 | 100.0 |
| 10 | 245.55 | 5.14 | 100.0 |
| 15 | 210.12 | 5.00 | 100.0 |
| 20 | 194.54 | 4.85 | 100.0 |
| 30 | 155.19 | 4.57 | 100.0 |

### 4.2.4 Promedio por columna especificada

Para el cálculo del promedio por columna especificada, respeto a la frecuencia ponderada, se utilizarán el conteo por columna especificada y la suma por columna especificada, en ambos casos la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, se tendrán que calcular ambos resultados, una vez calculados éstos, se dividirá la suma entre el conteo, con este cálculo se obtendrá el promedio, la consulta entonces queda de la siguiente manera:

select L_SUPPKEY, ( sum( l_extendedprice) + coalesce( ( suma * 1.0 * ( count( l_extendedprice) * 1.0 / total)), 0)) / ( count( l_extendedprice) + ( contador * 1.0 * ( count( l_extendedprice) * 1.0 / total))) as fw_avg from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select sum( l_extendedprice) as suma, count( l_extendedprice) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, suma, contador, total;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.25: Promedio por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 333.80 | 5.94 | 100.0 |
| 10 | 311.26 | 5.77 | 100.0 |
| 15 | 278.34 | 5.61 | 100.0 |
| 20 | 250.27 | 5.44 | 100.0 |
| 30 | 199.37 | 5.11 | 100.0 |

#### Promedio total por columna especificada

Para el cálculo del promedio total por columna especificada respecto a la frecuencia ponderada, se calcularán primero el conteo total por columna especificada y la suma total por columna especificada, en ambos casos la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, una vez calculadas estas dos consultas se tendrá que dividir la suma total entre el conteo total, con este cálculo se obtendrá el promedio total, la consulta entonces queda de la siguiente manera:

select sum( fw_sum) / sum( fw_count) as fw_avg_total from ( select L_SUPPKEY, sum( l_extendedprice) + coalesce( ( suma * 1.0 * ( count( l_extendedprice) * 1.0 / total)), 0) as fw_sum, count( l_extendedprice) + ( contador * 1.0 * ( count( l_extendedprice) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select sum( l_extendedprice) as suma, count( l_extendedprice) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, suma, total) as sub3;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.26: Promedio total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 329.39 | 5.72 | 100.0 |
| 10 | 303.36 | 5.56 | 100.0 |
| 15 | 282.33 | 5.39 | 100.0 |
| 20 | 249.97 | 5.23 | 100.0 |
| 30 | 189.87 | 4.90 | 100.0 |

### 4.2.5 Valor máximo por columna especificada

Para el cálculo del valor máximo por columna especificada respecto a la frecuencia ponderada, la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, además se utilizarán otras variables que están contenidos en la tabla *tm_fw*, aquí los valores ya están agrupados respecto a la columna *L_SUPPKEY* con una referencia hacia llave foránea válida, y las variables que se utilizarán ya están calculadas. PostgreSQL a comparación de SciDB si soporta los cálculos matemáticos necesarios, se creará una función que contendrá estos cálculos y se mandara a llamar mediante la instrucción "SELECT fw_max() AS answer;", entonces la función queda interpretada de la siguiente manera:

```
create function fw_max() RETURNS void AS $$

declare

    varx numeric :=0;
    varr numeric :=1;
    ind record;
    n record;
    a float8 :=0;
    b float8 :=0;

begin

    for ind in select * from tm_fw limit 100 LOOP

        if (ind.e-1)=((ind.e-1)-(ind.m-1)) then

            for n in select * from tm_ae_max where s<=(ind.e-ind.m-1) order by
            s asc LOOP

                if(n.l_extendedprice<=ind.maxxx) then

                    varx:=varx+(ind.maxxx*((ind.e-ind.m-1)-n.s+1));
                    exit;
                else

                    varx:=varx+n.l_extendedprice;
                end if;
            END LOOP;
```

```
else

    a:=((ind.m-1)::int!);
    while a<1e+200 and b<(ind.m-2) loop

        a:=a*(ind.e-b);
        b:=b+1;
    end loop;
    varr:=varr*(((ind.e-1)-(ind.m-1)+1)::float/a);
    a:=1;
    for i in ((ind.e-1)-(ind.m-1)+2)..(ind.e-1) LOOP

        while a<1e+1 and b<(ind.m-2) and varr>7e+290 loop

            a:=a*(ind.e-b);
            b:=b+1;
        end loop;
        varr:=varr*i/a;
        a:=1;
    END LOOP;
    varx:=varx+((select l_extendedprice from tm_ae_max where s=1)*varr);
    for n in select * from tm_ae_max where s>=2 and s<=(ind.e-ind.m-1)
    order by s asc LOOP

        if varr>=3.47822214672238e-319 then

            varr:=varr::float/(ind.e-n.s+1)*((ind.e-n.s)-(ind.m-1)+1);
        end if;
        if(n.l_extendedprice<=ind.maxxx) then

            varx:=varx+(ind.maxxx*varr);
        else

            varx:=varx+(n.l_extendedprice*varr);
        end if;
    END LOOP;
end if;
varr:=1;
varr:=varr*((ind.e-ind.m+1)::float/((ind.m::int!)));
for i in (ind.e-ind.m+2)..(ind.e-b) loop

    varr:=varr*i;
END LOOP;
b:=0;
```

```
        varx:=varx::float/varr;
        RAISE NOTICE 'l_suppkey =% maxxx = % fw_max= % hora=%',
        ind.l_suppkey,ind.maxxx,varx,timeofday();
        varx :=0;
        varr :=1;
    END LOOP;
end;
$$ LANGUAGE plpgsql;
```

Para el cálculo de esta Función de Agregación Extendida, PostgreSQL tiene una limitación en el tamaño de las variables y al intentar calcular el valor máximo por columna especificada con la tabla que tiene un 30% de error de integridad referencial, este sistema marca un error de tamaño de variables, por esto mismo se optó por hacer el análisis de los siguientes porcentajes: 5%, 10%, 15% y 20% de error de integridad referencial. También solo se analizarán los 100 primeros elementos, entonces, los resultados de la consulta anterior aplicada a cada una de las 4 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.27: Valor máximo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 3328.67 | 1.12 | 100.0 |
| 10 | 9045.48 | 1.31 | 100.0 |
| 15 | 15237.57 | 1.46 | 100.0 |
| 20 | 20100.95 | 1.54 | 100.0 |

### 4.2.6 Valor mínimo por columna especificada

Para el cálculo del valor mínimo por columna especificada respecto a la frecuencia ponderada, la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, además se utilizarán otras variables que están contenidos en la tabla *tm_fw*, la fórmula es muy parecida a la utilizada en el cálculo del valor máximo, e igual las variables que se utilizarán ya están calculadas en la tabla *tm_fw*, y de igual forma se creará una función que contendrá los cálculos matemáticos necesarios y se mandara a llamar mediante la instrucción "SELECT fw_min() AS answer;", entonces la función queda interpretada de la siguiente manera:

```
create function fw_min() RETURNS void AS $$

declare

    varx numeric :=0;
    varr numeric :=1;
    ind record;
    n record;
    a float8 :=0;
    b float8 :=0;
begin

    for ind in select * from tm_fw limit 100 LOOP

        if (ind.e-1)=((ind.e-1)-(ind.m-1)) then

            for n in select * from tm_ae_min where s<=(ind.e-ind.m-1) order by s
            asc LOOP

                if(n.l_extendedprice>=ind.minnn) then

                    varx:=varx+(ind.minnn*((ind.e-ind.m-1)-n.s+1));
                    exit;
                else

                    varx:=varx+n.l_extendedprice;
                end if;
            END LOOP;
        else
```

```
    a:=((ind.m-1)::int!);
    while a<1e+200 and b<(ind.m-2) loop

        a:=a*(ind.e-b);
        b:=b+1;
    end loop;
    varr:=varr*(((ind.e-1)-(ind.m-1)+1)::float/a);
    a:=1;
    for i in ((ind.e-1)-(ind.m-1)+2)..(ind.e-1) LOOP

        while a<1e+1 and b<(ind.m-2) and varr>7e+290 loop

            a:=a*(ind.e-b);
            b:=b+1;
        end loop;
        varr:=varr*i/a;
        a:=1;
    END LOOP;
    varx:=varx+((select l_extendedprice from tm_ae_min where s=1)*varr);
    for n in select * from tm_ae_min_5 where s>=2 and s<=
    (ind.e-ind.m-1) order by s asc LOOP

        if varr>=3.47822214672238e-319 then

            varr:=varr::float/(ind.e-n.s+1)*((ind.e-n.s)-(ind.m-1)+1);
        end if;
        if(n.l_extendedprice>=ind.minnn) then

            varx:=varx+(ind.minnn*varr);
        else

            varx:=varx+(n.l_extendedprice*varr);
        end if;
    END LOOP;
end if;
varr:=1;
varr:=varr*((ind.e-ind.m+1)::float/((ind.m::int!)));
for i in (ind.e-ind.m+2)..(ind.e-b) loop

    varr:=varr*i;
END LOOP;
b:=0;
varx:=varx::float/varr;
RAISE NOTICE 'l_suppkey =% minnn = % fw_min= % hora=%',
```

```
        ind.l_suppkey,ind.minnn,varx,timeofday();
        varx :=0;
        varr :=1;
    END LOOP;
end;
$$ LANGUAGE plpgsql;
```

Al igual que en el cálculo del Valor máximo por columna especificada, para el cálculo de esta Función de Agregación Extendida, se tuvo que delimitar el porcentaje de error máximo a calcular en este caso al igual que en el anterior el límite es 20% de error de integridad referencial y también sólo se analizarán los 100 primeros elementos, entonces, los resultados de la consulta anterior aplicada a cada una de las 4 tablas correspondientes a cada porcentaje de error (5%, 10%, 15%, 20%) se muestran en la siguiente tabla:

Table 4.28: Valor mínimo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador postgres |
|---|---|---|---|
| 5 | 3335.36 | 1.12 | 100.0 |
| 10 | 9183.63 | 1.33 | 100.0 |
| 15 | 15330.98 | 1.60 | 100.0 |
| 20 | 20040.96 | 1.76 | 100.0 |

## 4.3 Experimentos en HP Vertica

Además de las tablas proporcionadas por TPCH, importadas desde archivos CSV creados por PostgreSQL, se crearon tablas extras para ayudar a calcular las Funciones de Agregación Extendida. La primer tabla contiene todos los registros con errores de integridad referencial ordenados de mayor a menor. La segunda tabla tiene la misma estructura y datos que la primera, sólo que los datos están ordenados de menor a mayor. La tercer tabla contendrá los registros que tengan una referencia válida respecto a la llave foránea. La cuarta tabla contendrá agrupaciones de los datos respecto a la llave foránea *L_SUPPKEY* que tenga una referencia válida, esta última ayudará al cálculo del valor mínimo y máximo, todos estos grupos de tablas estarán correlacionados a un porcentaje de error de integridad referencial (5%, 10%, 15%, 20%, 30%).

Las tablas de resultados contienen las siguientes columnas:

1. % de error: es el porcentaje de registros que tienen errores de integridad referencial en la tabla referenciante
2. Tiempo de ejecución: es el tiempo que tarda la consulta en ejecutarse
3. Memoria usada:Es cantidad de memoria RAM utilizada por la aplicación.
4. % de procesador: Es el porcentaje de procesador que utiliza la consulta al ejecutarse

**Tabla LI_SUPPLIER**

La tabla *LI_SUPPLIER* es la que contiene las registros de las tablas *LINEITEM* que tienen una referencia válida en la columna de llave foránea *L_SUPPKEY* respecto a la tabla *SUPPLIER* en su columna de llave primaria *S_SUPPKEY*, para esto se utilizará una reunión natural entre las dos columnas antes mencionadas y de esta reunión sólo se tomarán los campos *L_SUPPKEY* de tipo entero y *L_EXTENDEDPRICE* de tipo flotante.

create table LI_SUPPLIER as select LINEITEM.L_SUPPKEY, L_EXTENDEDPRICE from SUPPLIER inner join LINEITEM on SUPPLIER.S_SUPPKEY = LINEITEM.L_SUPPKEY order by LINEITEM.L_SUPPKEY desc;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.29: Tabla LI_SUPPLIER

| % de error | Tiempo de ejecución(Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 35.23356 | 4.73 | 266.8 |
| 10 | 43.563882 | 5.23 | 307.5 |
| 15 | 34.897559 | 5.05 | 192.5 |
| 20 | 36.34625 | 5.21 | 303.3 |
| 30 | 33.695194 | 4.81 | 193.8 |

**Tabla tm_ae_max**

La tabla *tm_ae_max* contiene las registros de las tablas *LINEITEM* que tengan una referencia inválida en su llave foránea en la columna *L_SUPPKEY* respecto a la tabla *SUPPLIER* con la columna de llave primaria *S_SUPPKEY*, para esto se utilizará una reunión por la derecha en las columnas antes mencionadas, seleccionando las registros que en la columna *SUPPLIER.S_SUPPKEY* tengan un valor nulo, de esta reunión solo se tomará la columna *L_EXTENDEDPRICE* de tipo flotante ordenados de mayor a menor y se agregará una columna extra de nombre *s* que realizará la función de conteo de las registros.

create table tm_ae_max as select L_EXTENDEDPRICE, row_number() over() as s from (select L_EXTENDEDPRICE from SUPPLIER right join LINEITEM on SUPPLIER.S_SUPPKEY = LINEITEM.L_SUPPKEY where LINEITEM.L_SUPPKEY is null order by L_EXTENDEDPRICE desc) as sub1;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.30: Tabla tm_ae_max

| % de error | Tiempo de ejecución(Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 16.368269 | 3.95 | 31.3 |
| 10 | 5.203454 | 3.87 | 38.7 |
| 15 | 21.246399 | 3.95 | 66.8 |
| 20 | 19.509196 | 4.24 | 93.4 |
| 30 | 22.002429 | 4.38 | 108.7 |

**Tabla tm_ae_min**

La tabla *tm_ae_min* es básicamente igual a la tabla *tm_ae_max* creada arriba, tanto en datos como en estructura, la diferencia es que los datos de la columna *L_EXTENDEDPRICE* se ordenan de menor a mayor, y se crea un nuevo conteo con el mismo nombre *s*. Así que se utilizó la tabla *tm_ae_max* como base para la consulta, se ordenará y después se creará la columna *s*.

create table tm_ae_min as select L_EXTENDEDPRICE, row_number() over() as s from (select l_extendedprice from tm_ae_max order by l_extendedprice asc)as sub1;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.31: Tabla tm_ae_min

| % de error | Tiempo de ejecución(Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 1.135664 | 3.19 | 13.3 |
| 10 | 1.26448 | 3.33 | 18.6 |
| 15 | 1.655962 | 3.34 | 38.9 |
| 20 | 1.868084 | 3.62 | 52.2 |
| 30 | 1.877521 | 4.08 | 82.5 |

**Tabla tm_fw**

La última tabla en crearse es la tabla *tm_fw* esta es la tabla auxiliar que ayudará en el cálculo de las Funciones de Agregación Extendida, para encontrar el valor máximo y mínimo, esta tabla tiene los valores de la tabla *LI_SUPPLIER* agrupados por la columna *L_SUPPKEY*, los datos que se calcularán son los necesarios para las Funciones de Agregación Extendida.

create table tm_fw as select L_SUPPKEY, count( l_extendedprice) * 1.0 / t1 as pk, ceil( ( count( l_extendedprice) * 1.0 / t1) * e) as m, e, max( l_extendedprice) as maxxx, min( l_extendedprice) as minnn from LI_SUPPLIER cross join ( select count( l_extendedprice) as t1 from LI_SUPPLIER) as sub1 cross join (select count(l_extendedprice) as e from tm_ae_max) as sub2 group by L_SUPPKEY, e, t1 order by L_SUPPKEY;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

En esta primera parte se tendrían las tablas auxiliares, que separan los datos con errores de integridad referencial de los datos que no tienen errores de integridad referencial. En las siguientes implementaciones de las Funciones de Agregación

Table 4.32: Tabla tm_fw

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 11.489137 | 3.43 | 420.5 |
| 10 | 10.348037 | 3.44 | 438.4 |
| 15 | 11.258885 | 3.41 | 427.3 |
| 20 | 9.269148 | 3.40 | 470.0 |
| 30 | 9.389273 | 3.37 | 431.0 |

Extendida se tendrá que tomar en cuenta que las tablas que se utilizan para estos cálculos ya están creadas, y se registraron los datos de las mismas.

### 4.3.1 Conteo por llave primaria

El conteo en las Funciones de Agregación Extendida utilizando la frecuencia ponderada, utiliza las tablas *LI_SUPPLIER* respecto ala columna *L_SUPPKEY* y la tabla *tm_ae_max*, la consulta es prácticamente igual a la ejecutada en PostgreSQL sólo que al final se agrega un ordenamiento por la columna *L_SUPPKEY*, entonces la consulta queda de la siguiente manera:

select L_SUPPKEY, count( L_SUPPKEY) + ( contador * 1.0 * ( count( L_SUPPKEY) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count(*) as total from LI_SUPPLIER) as sub1 cross join (select count(*) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total order by L_SUPPKEY;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.33: Conteo por llave primaria

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 8.226717 | 3.22 | 476.7 |
| 10 | 6.263881 | 3.16 | 444.5 |
| 15 | 6.56616 | 3.16 | 421.0 |
| 20 | 5.525485 | 3.15 | 388.6 |
| 30 | 4.814289 | 3.14 | 341.8 |

#### Conteo total por llave primaria

Para realizar el conteo total por llave primaria se utilizará la consulta anterior (sin el ordenamiento ya que para este conteo no impacta de ninguna manera) a la cual se le agregará una sumatoria a todos los resultados mostrados en la columna *fw_count*, con esto queda una consulta igual a la ejecutada en PostgreSQL, la consulta entonces queda de la siguiente forma:

select sum( fw_count) as fw_count_total from ( select L_SUPPKEY, count( L_SUPPKEY) + (contador * 1.0 * ( count( L_SUPPKEY) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count(*) as total from LI_SUPPLIER) as sub1 cross join (select count(*) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total) as sub3;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas corres-

pondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.34: Conteo total por llave primaria

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 3.608836 | 3.17 | 423.2 |
| 10 | 3.750058 | 3.17 | 427.6 |
| 15 | 3.15695 | 3.16 | 406.1 |
| 20 | 3.082976 | 3.16 | 389.6 |
| 30 | 2.791633 | 3.16 | 343.2 |

### 4.3.2 Conteo por columna especificada

Para el cálculo del conteo por columna especificada, la columna escogida, respecto a la frecuencia ponderada es *L_EXTENDEDPRICE*, la consulta es muy parecida a la anterior solo que en vez de utilizar la columna llave se ocupa cualquier otra columna, entonces se modificará la consulta anterior para realizar los cálculos sobre la columna *L_EXTENDEDPRICE* en la tabla *LI_SUPPLIER*, la consulta es prácticamente igual a la ejecutada en PostgreSQL solo que al final se agrega un ordenamiento por la columna *L_SUPPKEY*, la consulta entonces queda de la siguiente manera:

select L_SUPPKEY, count( l_extendedprice) + ( contador * 1.0 * ( count( l_extendedprice) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select count( l_extendedprice) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total order by L_SUPPKEY;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.35: Conteo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 15.002809 | 3.41 | 537.5 |
| 10 | 14.23709 | 3.40 | 527.8 |
| 15 | 14.12379 | 3.38 | 377.6 |
| 20 | 13.580568 | 3.37 | 381.9 |
| 30 | 12.345105 | 3.34 | 416.4 |

#### Conteo total por columna especificada

Para realizar el conteo total por columna especificada, que en este caso se realiza sobre la columna *L_EXTENDEDPRICE*, se tomará como base la consulta anterior (sin el ordenamiento ya que para este conteo no impacta de ninguna manera) y se le agregará una sumatoria a todos los datos mostrados en la columna *fw_count*, con esto queda una consulta igual a la ejecutada en PostgreSQL, la consulta queda entonces de la siguiente manera:

select sum( fw_count) as fw_count_total from ( select L_SUPPKEY, count( l_extendedprice) + (contador * 1.0 * ( count( l_extendedprice) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join (select count( l_extendedprice) as contador

from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total) as sub3;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.36: Conteo total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 11.077072 | 3.42 | 535.3 |
| 10 | 10.303796 | 3.40 | 435.8 |
| 15 | 10.499575 | 3.39 | 347.5 |
| 20 | 8.951291 | 3.38 | 381.6 |
| 30 | 9.529521 | 3.35 | 398.2 |

### 4.3.3 Suma por columna especificada

Para el cálculo de la suma por columna especificada, respecto a la frecuencia ponderada, se utilizará como base la columna *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, la consulta es prácticamente igual a la ejecutada en PostgreSQL solo que al final se agrega un ordenamiento por la columna *L_SUPPKEY*, la consulta entonces queda de la siguiente manera:

select L_SUPPKEY, sum( l_extendedprice) + coalesce( ( suma * 1.0 * ( count( l_extendedprice) * 1.0 / total)), 0) as fw_sum from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select sum( l_extendedprice) as suma from tm_ae_max) as sub2 group by L_SUPPKEY, suma, total order by L_SUPPKEY;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.37: Suma por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 13.950383 | 3.41 | 375.3 |
| 10 | 13.702238 | 3.41 | 431.2 |
| 15 | 14.701287 | 3.38 | 469.7 |
| 20 | 13.120788 | 3.37 | 391.3 |
| 30 | 12.074031 | 3.34 | 416.5 |

**Suma total por columna especificada**

Para el cálculo de la suma total por columna especificada, que en este caso se utiliza la columna *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* se toma como base la consulta anterior (sin el ordenamiento ya que para este conteo no impacta de ninguna manera) agregándole una sumatoria de a todos los valores mostrados en la columna *fw_sum*, con esto queda una consulta igual a la ejecutada en PostgreSQL, la consulta queda entonces de la siguiente manera:

select sum( fw_sum) as fw_sum_total from( select L_SUPPKEY, sum( l_extendedprice) + coalesce( ( suma * 1.0 *( count( l_extendedprice) * 1.0 / total)), 0) as fw_sum from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select sum( l_extendedprice) as suma from tm_ae_max) as sub2 group by L_SUPPKEY, suma, total)as sub3;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.38: Suma total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 10.895649 | 3.42 | 456.7 |
| 10 | 9.979282 | 3.41 | 402.7 |
| 15 | 10.053139 | 3.39 | 461.4 |
| 20 | 9.980012 | 3.38 | 464.4 |
| 30 | 9.047905 | 3.35 | 423.8 |

### 4.3.4 Promedio por columna especificada

Para el cálculo del promedio por columna especificada, respeto a la frecuencia ponderada, se utilizarán dos funciones el conteo por columna especificada y la suma por columna especificada, en ambos casos la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, se tendrán que calcular ambos resultados, una vez calculados estos, se dividirá la suma entre el conteo, con este cálculo se obtendrá el promedio. La consulta es prácticamente igual a la ejecutada en PostgreSQL solo que al final se agrega un ordenamiento en la columna *L_SUPPKEY*, la consulta entonces queda de la siguiente manera:

select L_SUPPKEY, ( sum( l_extendedprice) + coalesce( ( suma * 1.0 * ( count( l_extendedprice) * 1.0 / total)), 0)) / ( count( l_extendedprice) + ( contador * 1.0 * ( count( l_extendedprice) * 1.0 / total))) as fw_avg from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join (select sum( l_extendedprice) as suma, count( l_extendedprice) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, suma, contador, total order by L_SUPPKEY;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.39: Promedio por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 15.347455 | 3.41 | 539.6 |
| 10 | 13.013918 | 3.41 | 522.9 |
| 15 | 14.103408 | 3.40 | 527.2 |
| 20 | 12.536048 | 3.38 | 444.1 |
| 30 | 12.261818 | 3.36 | 476.0 |

#### Promedio total por columna especificada

Para el cálculo del promedio total por columna especificada respecto a la frecuencia ponderada, se calcularán primero el conteo total por columna especificada y la suma total por columna especificada( sin el ordenamiento ya que para este conteo no impacta de ninguna manera), en ambos casos la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, una vez calculadas estas dos consultas se tendrá que dividir la suma total entre el conteo total, con esto se obtendrá el promedio total, con esto queda una consulta igual a la ejecutada en PostgreSQL, la consulta

entonces queda de la siguiente manera:

select sum( fw_sum) / sum( fw_count) as fw_avg_total from ( select L_SUPPKEY, sum( l_extendedprice) + coalesce( ( suma * 1.0 * ( count( l_extendedprice) * 1.0 / total)), 0) as fw_sum, count( l_extendedprice) + ( contador * 1.0 * ( count( l_extendedprice) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join ( select sum( l_extendedprice) as suma, count( l_extendedprice) as contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, suma, total) as sub3;

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.40: Promedio total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador vertica |
|---|---|---|---|
| 5 | 11.067113 | 3.42 | 411.3 |
| 10 | 11.002556 | 3.41 | 449.1 |
| 15 | 10.875698 | 3.40 | 487.0 |
| 20 | 9.628331 | 3.39 | 395.9 |
| 30 | 9.049158 | 3.35 | 468.5 |

### 4.3.5 Valor máximo por columna especificada

Para el cálculo del valor máximo por columna especificada, respecto a la frecuencia ponderada, la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, además se utilizarán otras variables que están contenidas en la tabla *tm_fw*, aquí los valores ya están agrupados respecto a la columna *L_SUPPKEY* con una referencia válida, y las variables que se utilizarán ya están calculadas, debido a que esta función requiere muchos cálculos matemáticos que no soporta internamente HP Vertica se optó por utilizar una alternativa recomendada, Python, se utilizó el ODBC de Python para la conexión con HP Vertica, se configuraron las variables de conexión correspondientes, y así la ejecución del código en Python quedo de la siguiente manera:

```
import pyodbc
import math
from decimal import *
import time
time.strftime("%a, %d %b %Y %H:%M:%S +0000")
db = pyodbc.connect("DSN=VerticaDSN")
cur = db.cursor()
cur.execute("select * from tm_fw limit 100")
x=cur.fetchall()
cur.execute("select * from tm_ae_max where s>=2 order by s asc")
y=cur.fetchall()
cur.execute("select * from tm_ae_max order by s asc")
z=cur.fetchall()
cur.execute("select L_EXTENDEDPRICE from tm_ae_max where s=1")
w=cur.fetchone()

for x1 in x:

    varx=Decimal(0)
    varr=Decimal(1)
    a=Decimal(0)
    b=Decimal(0)
    ee=Decimal(x1.e)
    mm=Decimal(x1.m)
    fac=math.factorial(mm-1)
    if ((ee-1)==((ee-1)-(mm-1))):

        for z1 in z:

            if (z1.s<=(ee-mm-1)):
```

```
            if z1.L_EXTENDEDPRICE<=x1.maxxx:

                varx=varx+(x1.maxxx*((ee-mm-1)-z1.s+1))
                break
            else:

                varx=varx+z1.L_EXTENDEDPRICE
else:

    a=fac
    while (a<(1E+200) and b<(mm-2)):

        a=a*(ee-b)
        b=b+1
    varr=varr*(((ee-1)-(mm-1)+1)/a)
    a=1
    for i in xrange (((ee-1)-(mm-1)+2),(ee)):

        while (a<1E+1 and b<(mm-2) and varr>7E+290):

            a=a*(ee-b)
            b=b+1
        varr=varr*Decimal(i)/a
        a=1
    varx=varx+(Decimal(w.L_EXTENDEDPRICE)*varr)
    for y1 in y:

        if varr>=3.47822214672238E-319:

            varr=varr/(ee-y1.s+1)*((ee-y1.s)-(mm-1)+1)
        if (y1.s<=(ee-mm-1)):

            if y1.L_EXTENDEDPRICE<=x1.maxxx:

                varx=varx+(Decimal(x1.maxxx)*varr)
            else:

                varx=varx+(Decimal(y1.L_EXTENDEDPRICE)*varr)
varr=1
varr=varr*((ee-mm+1)/(math.factorial(mm)))
for i in xrange((ee-mm+2),(ee-b+1)):

    varr=varr*Decimal(i)
varx=varx/varr
```

```
print('ind= %s maxxx= %s fw= %s' % (x1.L_SUPPKEY, x1.maxxx,varx))
time.strftime("%a, %d %b %Y %H:%M:%S +0000")
```

Para el cálculo de esta Función de Agregación Extendida se seguirá utilizando como límite máximo el 20% de error de integridad referencial, y de igual modo solo se analizarán los 100 primeros elementos, entonces, los resultados de la consulta anterior aplicada a cada una de las 4 tablas correspondientes a cada porcentaje de error (5%, 10%, 15%, 20%) se muestra en la siguiente tabla:

Table 4.41: Valor máximo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | |
|---|---|---|---|---|
| | | | vertica | python |
| 5 | 75842 | 9.08 | 93.5 | 100.1 |
| 10 | 156925 | 13.94 | 102.7 | 100.1 |
| 15 | 224907 | 15.06 | 100.4 | 100.1 |
| 20 | 308355 | 15.50 | 126.3 | 100.1 |

### 4.3.6 Valor mínimo por columna especificada

Para el cálculo del valor mínimo por columna especificada, respecto a la frecuencia ponderada, la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, además se utilizarán otras variables que están contenidas en la tabla *tm_fw*, la fórmula es muy parecida a la utilizada en el cálculo del valor máximo, igual se iniciará con el código en Python que contendrá la conexión a HP Vertica y después se realizarán los cálculos matemáticos necesarios, entonces, la función queda interpretada de la siguiente manera:

```
import pyodbc
import math
from decimal import *
import time
time.strftime("%a, %d %b %Y %H:%M:%S +0000")
db = pyodbc.connect("DSN=VerticaDSN")
cur = db.cursor()
cur.execute("select * from tm_fw limit 100")
x=cur.fetchall()
cur.execute("select * from tm_ae_min where s¿=2 order by s asc")
y=cur.fetchall()
cur.execute("select * from tm_ae_min order by s asc")
z=cur.fetchall()
cur.execute("select L_EXTENDEDPRICE from tm_ae_min where s=1")
w=cur.fetchone()

for x1 in x:
    varx=Decimal(0)

    varr=Decimal(1)
    a=Decimal(0)
    b=Decimal(0)
    ee=Decimal(x1.e)
    mm=Decimal(x1.m)
    fac=math.factorial(mm-1)
    if ((ee-1)==((ee-1)-(mm-1))):

        for z1 in z:

            if (z1.s<=(ee-mm-1)):
```

```
            if z1.L_EXTENDEDPRICE>=x1.minnn:

                varx=varx+(x1.minnn*((ee-mm-1)-z1.s+1))
                break
            else:

                varx=varx+z1.L_EXTENDEDPRICE
else:

    a=fac
    while (a<(1E+200) and b<(mm-2)):

        a=a*(ee-b)
        b=b+1
    varr=varr*(((ee-1)-(mm-1)+1)/a)
    a=1
    for i in xrange (((ee-1)-(mm-1)+2),(ee)):

        while (a<1E+1 and b<(mm-2) and varr>7E+290):

            a=a*(ee-b)
            b=b+1
        varr=varr*Decimal(i)/a
        a=1
    varx=varx+(Decimal(w.L_EXTENDEDPRICE)*varr)
    for y1 in y:

        if (y1.s<=(ee-mm-1)):

            if varr>=3.47822214672238E-319:

                varr=varr/(ee-y1.s+1)*((ee-y1.s)-(mm-1)+1)
            if y1.L_EXTENDEDPRICE>=x1.minnn:

                varx=varx+(Decimal(x1.minnn)*varr)
            else:

                varx=varx+(Decimal(y1.L_EXTENDEDPRICE)*varr)
varr=1
varr=varr*((ee-mm+1)/(math.factorial(mm)))
for i in xrange((ee-mm+2),(ee-b+1)):

    varr=varr*Decimal(i)
varx=varx/varr
```

```
print('ind= %s maxxx= %s fw= %s' % (x1.L_SUPPKEY, x1.maxxx,varx))
time.strftime("%a, %d %b %Y %H:%M:%S +0000")
```

Al igual que en el cálculo del Valor máximo por columna especificada, para el cálculo de esta Función de Agregación Extendida se seguirá utilizando como límite máximo el 20% de error de integridad referencial y de igual modo solo se analizarán los 100 primeros elementos, entonces, los resultados de la consulta anterior aplicada a cada una de las 4 tablas correspondientes a cada porcentaje de error (5%, 10%, 15%) se muestran en la siguiente tabla:

Table 4.42: Valor mínimo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador | |
|---|---|---|---|---|
| | | | vertica | python |
| 5 | 77536 | 8.85 | 100.4 | 100.1 |
| 10 | 158392 | 13.51 | 101.5 | 100.1 |
| 15 | 229320 | 14.37 | 113.7 | 100.1 |

## 4.4 Experimentos en Apache Spark

Con los datos cargados en las vistas temporales representando a las tablas a utilizar, se ejecutarán las consultas y los resultados serán guardados en DataFrames, para después crear nuevas tablas que se guardarán en archivos con extensión '.parquet' y cuando se necesiten se guardarán en DataFrames para posteriormente crearse vistas temporales. Las tablas extras ayudarán en el cálculo de las Funciones de Agregación Extendida. La primer tabla contendrá todos los registros con errores de integridad referencial ordenados de mayor a menor. La segunda tabla tendrá la misma estructura y datos que la anterior sólo que los datos estarán ordenados de menor a mayor. La tercer tabla contendrá los registros que tengan una referencia válida respecto a la llave foránea. La cuarta tabla contendrá los registros con una referencia válida respecto a la llave foránea agrupando los datos por los valores de llave foránea *L_SUPPKEY*. Esta última tabla ayudará al cálculo del valor mínimo y máximo, todos estos grupos de tablas estarán correlacionados a un porcentaje de error de integridad referencial (5%, 10%, 15%, 20%, 30%).

Las tablas de resultados contienen las siguientes columnas:

1. % de error: es el porcentaje de registros que tienen errores de integridad referencial en la tabla referenciante
2. Tiempo de ejecución: es el tiempo que tarda la consulta en ejecutarse
3. Memoria usada:Es cantidad de memoria RAM utilizada por la aplicación.
4. % de procesador: Es el porcentaje de procesador que utiliza la consulta al ejecutarse

**Tabla LI_SUPPLIER**

La tabla *LI_SUPPLIER* es la que contiene las registros de las tablas *LINEITEM* que tengan una referencia válida en la columna de llave foránea *L_SUPPKEY* respecto a la tabla *SUPPLIER* en su columna de llave primaria *S_SUPPKEY*, para esto se utilizará una reunión natural entre las dos columnas antes mencionadas y de esta reunión solo se tomarán los campos *L_SUPPKEY* de tipo entero y *L_EXTENDEDPRICE*, esto se pondrá dentro de una función y los resultados de la consulta los pondrá en un DataFrame para después guardarlo en un archivo con extensión *.parquet*, cuando se quierá utilizar el contenido del archivo se leerá y a partir de este se creará una vista temporal.

```
def ti_LI_SUPPLIER(){
    val start=System.nanoTime()
    val LI_SUPPLIERDF = spark.sql("select LINEITEM.L_SUPPKEY,
    L_EXTENDEDPRICE from SUPPLIER inner join LINEITEM
```

```
    on SUPPLIER.S_SUPPKEY = LINEITEM.L_SUPPKEY order
    by LINEITEM.L_SUPPKEY desc")
    LI_SUPPLIERDF.write.parquet("LI_SUPPLIER.parquet")
    val LI_SUPPLIER2DF = spark.read.parquet("LI_SUPPLIER.parquet")
    LI_SUPPLIER2DF.createOrReplaceTempView("LI_SUPPLIER")
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Una vez creada la función sólo se tiene que llamar con la siguiente linea:

ti_LI_SUPPLIER()

Los resultados de la consulta anterior contenida en la función aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.43: Tabla LI_SUPPLIER

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB | %Procesador | |
|---|---|---|---|---|
| | | usada GB | java | postgres |
| 5 | 329.28 | 15.52 | 599.7 | 99.1 |
| 10 | 312.73 | 15.52 | 599.7 | 98.8 |
| 15 | 295.48 | 15.52 | 588.1 | 99.1 |
| 20 | 289.57 | 15.52 | 591.4 | 98.8 |
| 30 | 251.89 | 15.52 | 583.4 | 98.8 |

**Tabla tm_ae_max**

La tabla *tm_ae_max* contendrá las registros de las tablas *LINEITEM* que tengan una referencia inválida en su llave foránea en la columna *L_SUPPKEY*, respecto a la tabla *SUPPLIER* en la columna de llave primaria *S_SUPPKEY*, para esto se utilizará una reunión por la derecha entre las columnas antes mencionadas, tomando las registros que en la columna *SUPPLIER.S_SUPPKEY* tengan un valor nulo de esta reunión sólo se tomará la columna *L_EXTENDEDPRICE* con sus datos ordenados de mayor a menor y se agregará una columna extra, de nombre *s* que realizará la función de conteo de las registros iniciando en 1 y terminando en el número total de registros, esto se pondrá dentro de una función y los resultados de la consulta los pondrá en un DataFrame para después guardarlo en un archivo con extensión *.parquet*, cuando se quierá utilizar el contenido del archivo se leerá y a partir de este se creará una vista temporal.

```
def ti_tm_ae_max(){
```

```
    val start=System.nanoTime()
    val tm_ae_maxDF = spark.sql("select L_EXTENDEDPRICE,
    row_number() over(PARTITION BY 1 ORDER BY 1 ) as s
    from (select L_EXTENDEDPRICE from SUPPLIER right
    join LINEITEM on SUPPLIER.S_SUPPKEY =
    LINEITEM.L_SUPPKEY where LINEITEM.L_SUPPKEY is
    null order by L_EXTENDEDPRICE desc) as sub1")
    tm_ae_maxDF.write.parquet("tm_ae_max.parquet")
    val tm_ae_max2DF = spark.read.parquet("tm_ae_max.parquet")
    tm_ae_max2DF.createOrReplaceTempView("tm_ae_max")
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Una vez creada la función solo se tiene que llamar con la siguiente línea:

```
ti_tm_ae_max()
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.44: Tabla tm_ae_max

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | %Procesador java | %Procesador postgres |
|---|---|---|---|---|
| 5 | 124.84 | 11.77 | 269.3 | 97.8 |
| 10 | 138.81 | 11.85 | 347.4 | 93.1 |
| 15 | 155.35 | 12.75 | 347.4 | 93.4 |
| 20 | 169.98 | 12.96 | 364.7 | 95.1 |
| 30 | 195.53 | 14.56 | 395.9 | 92.1 |

**Tabla tm_ae_min**

La tabla *tm_ae_min* es básicamente igual a la tabla *tm_ae_max* creada con anterioridad, tanto en datos como en estructura, la diferencia es, que los datos de la columna *L_EXTENDEDPRICE* en este caso se ordenan de menor a mayor, y se crea un nuevo conteo con el nombre *s*, así que se utilizó la tabla *tm_ae_max* como base para esta consulta, se ordenarán los datos y después se creará la columna *s*, esto se pondrá dentro de una función y los resultados de la consulta los pondrá en un DataFrame para después guardarlo en un archivo con extensión *.parquet*, cuando se quierá utilizar el contenido del archivo se leerá y a partir de este se creará una vista temporal.

```
def ti_tm_ae_min(){
    val start=System.nanoTime()
    val tm_ae_min_5DF = spark.sql("select L_EXTENDEDPRICE, row_number()
    over(PARTITION BY 1 ORDER BY 1 ) as s from (select l_extendedprice
    from tm_ae_max order by l_extendedprice asc)as sub1")
    tm_ae_min_5DF.write.parquet("tm_ae_min.parquet")
    val tm_ae_min_52DF = spark.read.parquet("tm_ae_min.parquet")
    tm_ae_min_52DF.createOrReplaceTempView("tm_ae_min")
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Una vez creada la función solo se tiene que llamar con la siguiente línea:

```
ti_tm_ae_min()
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.45: Tabla tm_ae_min

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | %Procesador java |
|---|---|---|---|
| 5 | 16.61 | 3.02 | 441.5 |
| 10 | 22.31 | 3.62 | 474.6 |
| 15 | 26.53 | 3.46 | 451.8 |
| 20 | 33.25 | 5.33 | 531.9 |
| 30 | 43.25 | 6.37 | 552.1 |

**Tabla tm_fw**

La última tabla es *tm_fw* esta contendra las registros con una referencia válida agrupadas por la culumna *L_SUPPKEY*, esta tabla servirá para el cálculo de las funciones max() y min(), esto se pondrá dentro de una función y los resultados de la consulta los pondrá en un DataFrame para después guardarlo en un archivo con extensión *.parquet*, cuando se quierá utilizar el contenido del archivo se leerá y a partir de este se creará una vista temporal.

```
def ti_tm_fw(){
    val start=System.nanoTime()
    val tm_fwDF = spark.sql("select L_SUPPKEY, count( l_extendedprice) *
    1.0 / t1 as pk, ceil( ( count( l_extendedprice) * 1.0 / t1) * e) as m, e, max(
    l_extendedprice) as maxxx, min( l_extendedprice) as minnn from LI_SUPPLIER
```

```
    cross join ( select count( l_extendedprice) as t1 from LI_SUPPLIER) as sub1
    cross join ( select count( l_extendedprice) as e from tm_ae_max) as sub2 group
    by L_SUPPKEY, e, t1 order by L_SUPPKEY")
    tm_fwDF.write.parquet("tm_fw.parquet")
    val tm_fw2DF = spark.read.parquet("tm_fw.parquet")
    tm_fw2DF.createOrReplaceTempView("tm_fw")
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Una vez creada la función solo se tiene que llamar con la siguiente línea:

```
ti_tm_fw()
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.46: Tabla tm_fw

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | %Procesador java |
|---|---|---|---|
| 5 | 42.07 | 6.15 | 576.1 |
| 10 | 42.25 | 5.23 | 565.1 |
| 15 | 39.67 | 5.59 | 567.4 |
| 20 | 39.36 | 4.87 | 560.4 |
| 30 | 36.97 | 4.96 | 572.1 |

En esta primera parte se tendrían las tablas auxiliares, que separan los datos con errores de integridad referencial de los datos que no tienen errores de integridad referencial. En las siguientes implementaciones de las Funciones de Agregación Extendida se tendrá que tomar en cuenta que las tablas que se utilizan para estos cálculos ya están creadas, y se registraron los datos de las mismas.

### 4.4.1 Conteo por llave primaria

El conteo, en las Funciones de Agregación Extendida, respecto a la frecuencia ponderada, utiliza las tablas *LI_SUPPLIER* con su columna *L_SUPPKEY* y *tm_ae_max*, la consulta es igual a la ejecutada en HP Vertica, sólo que los resultados se guardan en un DataFrame y se van imprimiendo línea por línea, hasta haber impreso todos los registros del DataFrame, todo esto se guarda en una función que queda de la siguiente manera:

```
def ti_fw_count(){
    val start=System.nanoTime()
    val fw_countDF = spark.sql("select L_SUPPKEY, count( L_SUPPKEY) +
    ( contador * ( count( L_SUPPKEY) * 1.0 / total)) as fw_count from LI_SUPPLIER
    cross join ( select count(*) as total from LI_SUPPLIER) as sub1 cross join
(select
    count(*) as contador from tm_ae_max) as sub2 group by L_SUPPKEY,
    contador, total order by L_SUPPKEY")
    fw_countDF.collect.foreach(println)
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.47: Conteo por llave primaria

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 31.17 | 3.14 | 561.8 |
| 10 | 31.17 | 3.26 | 557.5 |
| 15 | 31.04 | 3.08 | 550.1 |
| 20 | 31.54 | 3.38 | 552.2 |
| 30 | 29.66 | 3.11 | 530.9 |

#### Conteo total por llave primaria

Para realizar el conteo total por llave primaria, se utilizará la consulta anterior (sin el ordenamiento, ya que para este conteo no impacta de ninguna manera) a la cual se le agregará una sumatoria a todos los resultados mostrados en la columna

*fw_count*, la consulta es igual a la ejecutada en HP Vertica sólo que los resultados de la misma se guardan en un DataFrame y se van imprimiendo línea por línea hasta haber impreso todos los registros del DataFrame, todo esto se guarda en una función que queda de la siguiente manera:

```
def ti_fw_count_total(){
    val start=System.nanoTime()
    val fw_count_totalDF = spark.sql("select sum( fw_count) as fw_count_total
    from (select L_SUPPKEY, count( L_SUPPKEY) + ( contador * ( count
    ( L_SUPPKEY) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join (
select
    count(*) as total from LI_SUPPLIER) as sub1 cross join (select count(*) as
    contador from tm_ae_max) as sub2 group by L_SUPPKEY, contador, total)
    as sub3")
    fw_count_totalDF.collect.foreach(println)
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.48: Conteo total por llave primaria

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 20.81 | 3.03 | 557.5 |
| 10 | 21.45 | 2.96 | 551.5 |
| 15 | 21.49 | 2.91 | 549.9 |
| 20 | 21.03 | 3.00 | 518.2 |
| 30 | 19.76 | 3.07 | 537.7 |

### 4.4.2 Conteo por columna especificada

Para el cálculo del conteo por columna especificada, respecto a la frecuencia ponderada, la columna que se escogió para este conteo es *L_EXTENDEDPRICE*. Se modificará la consulta anterior para realizar los cálculos sobre esta columna en la tabla *LI_SUPPLIER*( en lugar de la columna de llave primaria), la consulta es igual a la ejecutada en HP Vertica sólo que los resultados de la misma se guardan en un DataFrame y se van imprimiendo línea por línea hasta haber impreso todos los registros del DataFrame, todo esto se guarda en una función que queda de la siguiente manera:

```
def ti_fw_count(){
    val start=System.nanoTime()
    val fw_countDF = spark.sql("select L_SUPPKEY, count( l_extendedprice) +
    ( contador * ( count( l_extendedprice) * 1.0 / total)) as fw_count from LI_SUPPLIER
    cross join ( select count( l_extendedprice) as total from LI_SUPPLIER) as
sub1
    cross join ( select count( l_extendedprice) as contador from tm_ae_max) as
    sub2 group by L_SUPPKEY, contador, total order by L_SUPPKEY")
    fw_countDF.collect.foreach(println)
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.49: Conteo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 41.79 | 4.97 | 569.9 |
| 10 | 43.52 | 5.04 | 570.8 |
| 15 | 41.30 | 4.67 | 566.1 |
| 20 | 40.17 | 4.52 | 572.5 |
| 30 | 40.48 | 4.67 | 569.0 |

#### Conteo total por columna especificada

Para realizar el conteo total por columna especificada, en este caso se hace sobre la columna *L_EXTENDEDPRICE*, se tomará como base la consulta anterior(sin el ordenamiento, ya que para este conteo no impacta de ninguna manera) y se le agregará una sumatoria en la columna *fw_count*, la consulta es igual a la ejecutada en HP Vertica sólo que los resultados de la misma se guardan en un DataFrame y se van

imprimiendo línea por línea hasta haber impreso todos los registros del DataFrame, todo esto se guarda en una función que queda de la siguiente manera:

```
def ti_fw_count_total(){
    val start=System.nanoTime()
    val fw_count_totalDF = spark.sql(" select sum( fw_count) as fw_count_total
    from (select L_SUPPKEY, count( l_extendedprice) + ( contador * ( count(
    l_extendedprice) * 1.0 / total)) as fw_count from LI_SUPPLIER cross join (
    select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join
    (select count( l_extendedprice) as contador from tm_ae_max) as sub2 group
    by L_SUPPKEY, contador, total) as sub3")
    fw_count_totalDF.collect.foreach(println)
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.50: Conteo total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 33.49 | 4.58 | 576.1 |
| 10 | 33.37 | 4.41 | 571.1 |
| 15 | 33.36 | 4.31 | 569.5 |
| 20 | 30.79 | 4.13 | 568.1 |
| 30 | 29.50 | 4.09 | 578.1 |

### 4.4.3 Suma por columna especificada

Para el cálculo de la suma por columna especificada, respecto a la frecuencia ponderada, se utilizará como base la columna *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, la consulta es igual a la ejecutada en HP Vertica sólo que los resultados de la misma se guardan en un DataFrame y se van imprimiendo línea por línea hasta haber impreso todos los registros del DataFrame, todo esto se guarda en una función que queda de la siguiente manera:

```
def ti_fw_sum(){
    val start=System.nanoTime()
    val fw_sumDF = spark.sql("select L_SUPPKEY, sum( l_extendedprice) +
    coalesce( ( suma * ( count( l_extendedprice) * 1.0 / total)), 0) as fw_sum
    from LI_SUPPLIER cross join ( select count( l_extendedprice) as total from
    LI_SUPPLIER) as sub1 cross join ( select sum( l_extendedprice) as suma
from
    tm_ae_max) as sub2 group by L_SUPPKEY, suma, total order by L_SUPPKEY")
    fw_sumDF.collect.foreach(println)
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.51: Suma por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 40.51 | 5.12 | 576.5 |
| 10 | 42.61 | 4.83 | 557.5 |
| 15 | 40.06 | 3.53 | 570.1 |
| 20 | 39.38 | 4.68 | 571.5 |
| 30 | 37.72 | 4.88 | 567.5 |

#### Suma total por columna especificada

Para el cálculo de la suma total por columna especificada, se utiliza la columna *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER*, se toma como base la consulta anterior(sin el ordenamiento, ya que para este conteo no impacta de ninguna manera) agregándole una sumatoria de los valores de la columna *fw_sum*, la consulta es igual a la ejecutada en HP Vertica sólo que los resultados de la misma se guardan en un DataFrame y se van imprimiendo línea por línea hasta haber impreso todos

los registros del DataFrame, todo esto se guarda en una función que queda de la siguiente manera:

```
def ti_fw_sum_total(){
    val start=System.nanoTime()
    val fw_sum_totalDF = spark.sql("select sum( fw_sum) as fw_sum_total from(
    select L_SUPPKEY, sum( l_extendedprice) + coalesce( ( suma * ( count(
    l_extendedprice) * 1.0 / total)), 0) as fw_sum from LI_SUPPLIER cross join
(
    select count( l_extendedprice) as total from LI_SUPPLIER) as sub1 cross join
(
    select sum( l_extendedprice) as suma from tm_ae_max) as sub2 group by
    L_SUPPKEY, suma, total) as sub3")
    fw_sum_total_5DF.collect.foreach(println)
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.52: Suma total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 33.66 | 4.65 | 570.1 |
| 10 | 33.48 | 4.44 | 560.5 |
| 15 | 33.86 | 4.23 | 563.8 |
| 20 | 30.82 | 4.28 | 558.7 |
| 30 | 30.64 | 3.59 | 560.8 |

### 4.4.4 Promedio por columna especificada

Para el cálculo del promedio por columna especificada, respeto a la frecuencia ponderada, se utilizarán dos funciones el conteo por columna especificada y la suma por columna especificada, en ambos casos la columna selecciona es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, se tendrán que ejecutar ambas consultas para obtener ambos resultados, una vez calculados éstos, se dividirá la suma entre el conteo, con este cálculo se obtendrá el promedio, la consulta es igual a la ejecutada en HP Vertica sólo que los resultados de la misma se guardan en un DataFrame y se van imprimiendo línea por línea hasta haber impreso todos los registros del DataFrame, todo esto se guarda en una función que queda de la siguiente manera:

```
def ti_fw_avg(){
    val start=System.nanoTime()
    val fw_avgDF = spark.sql("select L_SUPPKEY, ( sum( l_extendedprice) +
    coalesce( ( suma * ( count( l_extendedprice) * 1.0 / total)), 0) ) / ( count(
    l_extendedprice) + ( contador * ( count( l_extendedprice) * 1.0 / total)))
    as fw_avg from LI_SUPPLIER cross join ( select count( l_extendedprice) as
total
    from LI_SUPPLIER) as sub1 cross join ( select sum( l_extendedprice) as
suma,
    count( l_extendedprice) as contador from tm_ae_max) as sub2 group by
    L_SUPPKEY, suma, contador, total order by L_SUPPKEY")
    fw_avgDF.collect.foreach(println)
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.53: Promedio por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 44.06 | 5.20 | 572.8 |
| 10 | 41.67 | 4.83 | 565.2 |
| 15 | 43.14 | 5.14 | 565.8 |
| 20 | 41.69 | 4.83 | 570.5 |
| 30 | 39.38 | 4.57 | 567.5 |

**Promedio total por columna especificada**

Para el cálculo del promedio total por columna especificada, se calcularán; primero el conteo total por columna especificada y la suma total por columna especificada( sin el ordenamiento ya que para este conteo no impacta de ninguna manera), en ambos casos la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, una vez obtenidos los resultados de las dos consultas, se tendrá que dividir la suma total entre el conteo total, con esto se obtendrá el promedio total, la consulta es igual a la ejecutada en HP Vertica sólo que los resultados de la misma se guardan en un DataFrame y se van imprimiendo línea por línea hasta haber impreso todos los registros del DataFrame, todo esto se guarda en una función que queda de la siguiente manera:

```
def ti_fw_avg_total(){
    val start=System.nanoTime()
    val fw_avg_totalDF = spark.sql("select sum( fw_sum)/ sum( fw_count) as
    fw_avg_total from (select L_SUPPKEY, sum( l_extendedprice) + coalesce( (
    suma * ( count( l_extendedprice) * 1.0 / total)), 0) as fw_sum, count(
    l_extendedprice) + ( contador * ( count( l _extendedprice) * 1.0 / total))
    as fw_count from LI_SUPPLIER cross join ( select count( l_extendedprice) as
    total from LI_SUPPLIER) as sub1 cross join (select sum( l_extendedprice) as
    suma, count( l_extendedprice) as contador from tm_ae_max) as sub2 group
    by L_SUPPKEY, contador, suma, total) as sub3")
    fw_avg_total_5DF.collect.foreach(println)
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Los resultados de la consulta anterior aplicada a cada una de las 5 tablas correspondientes a cada porcentaje de error, se muestran en la siguiente tabla:

Table 4.54: Promedio total por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 34.05 | 4.46 | 566.2 |
| 10 | 35.70 | 3.39 | 562.4 |
| 15 | 33.11 | 4.05 | 571.7 |
| 20 | 31.82 | 4.10 | 569.5 |
| 30 | 29.29 | 4.04 | 566.8 |

### 4.4.5 Valor máximo por columna especificada

Al no contar, el sistema con una función propia para el cálculo del factorial se optó por la creación de una función que cumpliera este objetivo, quedando la función de la siguiente manera:

```
def factorial(n:Long):Double =
{
    if (n==0)
    return 1
    else
    return n * factorial(n-1)
}
```

Para el cálculo del valor máximo por columna especificada, respecto a la frecuencia ponderada, la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, además se utilizarán otras variables que están contenidos en la tabla *tm_fw*, aquí los valores ya están agrupados respecto a la columna *L_SUPPKEY* con una referencia válida, y las variables que se utilizarán ya están calculadas, todos los cálculos necesarios se guardan en una función que queda de la siguiente manera:

```
def ti_fw_max(){

    val start=System.nanoTime()
    val xDF = spark.sql("select * from tm_fw order by L_SUPPKEY limit 100")
    xDF.collect.foreach(x1 => {

        var varx: Double=0
        var varr: Double=1
        var a: Double=0
        var b: Int=0
        var ee: Long =x1.getLong(3)
        var mm: Long =x1.getDecimal(2).longValue()
        var fac: Double =factorial(mm-1)
        if ((ee-1)==((ee-1)-(mm-1))){

            val zDF = spark.sql("select * from tm_ae_max order by s asc")
            zDF.collect.foreach(z1 => {

                if (z1.getInt(1)<=(ee-mm-1)){

                    if (z1.getDouble(0)<=x1.getDouble(4)){
```

```
                varx=varx+(x1.getDouble(4)*((ee-mm-1)-z1.getInt(1)+1))
                break
            }
            else{

                varx=varx+z1.getDouble(0)
            }
        }
    })
}else{

    a=fac
    while (a<(1E+200) && b<(mm-2)){

        a=a*(ee-b)
        b=b+1
    }
    varr=varr*(((ee-1)-(mm-1)+1)/a)
    a=1
    val i=0
    for (i<-((ee-1)-(mm-1)+2) to (ee-1)){

        while (a<1E+1 && b<(mm-2) && varr>7E+290){

            a=a*(ee-b)
            b=b+1
        }
        varr=varr*i/a
        a=1
    }
    val wDF = spark.sql("select L_EXTENDEDPRICE from tm_ae_max
    where s=1")
    wDF.collect.foreach(w1 => {

        varx=varx+(w1.getDouble(0)*varr)
    })
    val yDF = spark.sql("select * from tm_ae_max where s>=2 order by
    s asc")
    yDF.collect.foreach(y1 => {

        if (varr>=3.47822214672238E-319){

            varr=varr/(ee-y1.getInt(1)+1)*((ee-y1.getInt(1))-(mm-1)+1)
        }
```

```
                if (y1.getInt(1)<=(ee-mm-1)){

                    if (y1.getDouble(0)<=x1.getDouble(4)){

                        varx=varx+(x1.getDouble(4)*varr)
                    }
                    else{

                        varx=varx+(y1.getDouble(0)*varr)
                    }
                }
            })
        }
        varr=1
        varr=varr*((ee-mm+1)/(factorial(mm)))
        val i=0
        for(i<- (ee-mm+2) to (ee-b)){

            varr=varr*i
        }
        varx=varx/varr
        println("ind= "+x1.getInt(0) + " maxxx=" +x1.getDouble(4)+" fw="+varx)
    })
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Para el cálculo de esta Función de Agregación Extendida se seguirá utilizando como límite máximo el 20% de error de integridad referencial y de igual modo sólo se analizarán los 100 primeros elementos, entonces, los resultados de la consulta anterior aplicada a cada una de las 4 tablas correspondientes a cada porcentaje de error (5%, 10%, 15%, 20%) se muestran en la siguiente tabla:

Table 4.55: Valor máximo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 414.70 | 7.05 | 341.8 |
| 10 | 753.80 | 8.57 | 276.7 |
| 15 | 1156.23 | 9.44 | 487.2 |
| 20 | 1623.41 | 11.46 | 564.8 |

### 4.4.6 Valor mínimo por columna especificada

Al no contar, el sistema con una función propia para el cálculo del factorial se optó por la creación de una función que cumpliera este objetivo, quedando la función de la siguiente manera:

```
def factorial(n:Long):Double =
{
    if (n==0)
    return 1
    else
    return n * factorial(n-1)
}
```

Para el cálculo del valor mínimo por columna especificada, respecto a la frecuencia ponderada, la columna escogida es *L_EXTENDEDPRICE* de la tabla *LI_SUPPLIER* y la tabla *tm_ae_max* con su columna *L_EXTENDEDPRICE*, además se utilizarán otras variables que están contenidos en la tabla *tm_fw*, la fórmula es muy parecida a la utilizada en el cálculo del valor máximo, todos los cálculos necesarios se guardan en una función que queda expresada de la siguiente manera:

```
def ti_fw_min(){

    val start=System.nanoTime()
    val xDF = spark.sql("select * from tm_fw order by L_SUPPKEY limit 100")
    xDF.collect.foreach(x1 => {

        var varx: Double=0
        var varr: Double=1
        var a: Double=0
        var b: Int=0
        var ee: Long =x1.getLong(3)
        var mm: Long =x1.getDecimal(2).longValue()
        var fac: Double =factorial(mm-1)
        if ((ee-1)==((ee-1)-(mm-1))){

            val zDF = spark.sql("select * from tm_ae_min order by s asc")
            zDF.collect.foreach(z1 => {

                if (z1.getInt(1)<=(ee-mm-1)){

                    if (z1.getDouble(0)>=x1.getDouble(5)){

                        varx=varx+(x1.getDouble(5)*((ee-mm-1)-z1.getInt(1)+1))
```

```
                    break
                }
                else{

                    varx=varx+z1.getDouble(0)
                }
            }
        })
    }else{

        a=fac
        while (a<(1E+200) && b<(mm-2)){

            a=a*(ee-b)
            b=b+1
        }
        varr=varr*(((ee-1)-(mm-1)+1)/a)
        a=1
        val i=0
        for (i<-((ee-1)-(mm-1)+2) to (ee-1)){

            while (a<1E+1 && b<(mm-2) && varr>7E+290){

                a=a*(ee-b)
                b=b+1
            }
            varr=varr*i/a
            a=1
        }
        val wDF = spark.sql("select L_EXTENDEDPRICE from tm_ae_min
where s=1")
        wDF.collect.foreach(w1 => {

            varx=varx+(w1.getDouble(0)*varr)
        })
        val yDF = spark.sql("select * from tm_ae_min where s>=2 order by s
asc")
        yDF.collect.foreach(y1 => {

            if (y1.getInt(1)<=(ee-mm-1)){

                if (varr>=3.47822214672238E-319){

                    varr=varr/(ee-y1.getInt(1)+1)*((ee-y1.getInt(1))-(mm-1)+1)
```

```
                    }
                    if (y1.getDouble(0)>=x1.getDouble(5)){

                        varx=varx+(x1.getDouble(5)*varr)
                    }
                    else{

                        varx=varx+(y1.getDouble(0)*varr)
                    }
                }
            })
        }
        varr=1
        varr=varr*((ee-mm+1)/(factorial(mm)))
        val i=0
        for(i<- (ee-mm+2) to (ee-b)){

            varr=varr*i
        }
        varx=varx/varr
        println("ind= "+x1.getInt(0) + " minnn=" +x1.getDouble(5)+" fw="+varx)
    })
    val end = System.nanoTime()
    println("Tiempo: " + (end-start)/1e+9 + " segundos")
}
```

Al igual que en el cálculo del Valor máximo por columna especificada, para el cálculo de esta Función de Agregación Extendida se seguirá utilizando como límite máximo el 20% de error de integridad referencial y de igual modo sólo se analizarán los 100 primeros elementos, entonces, los resultados de la consulta anterior aplicada a cada una de las 4 tablas correspondientes a cada porcentaje de error (5%, 10%, 15%, 20%) se muestran en la siguiente tabla:

Table 4.56: Valor mínimo por columna especificada

| % de error | Tiempo de ejecución (Seg.) | Memoria = 15.66 GB usada GB | % Procesador java |
|---|---|---|---|
| 5 | 407.38 | 7.33 | 368.2 |
| 10 | 762.20 | 8.10 | 555.2 |
| 15 | 1185.07 | 9.96 | 583.4 |
| 20 | 1581.56 | 11.89 | 595.1 |

## 4.5 Resultados

En esta sección se mostrarán los resultados de los 4 sistemas, al momento de ejecutar cada una de las Funciones de Agregación Extendida. Los resultados de la ejecución de cada sistema se dividirán en 4 secciones:

1. Tiempo de ejecución.

2. Cantidad de memoria RAM utilizada.

3. Porcentaje de procesador que utiliza cada consulta.

### 4.5.1 Tabla LI_SUPPLIER

Los resultados obtenidos al momento de crear la tabla *LI_SUPPLIER* son los que se muestran a continuación.

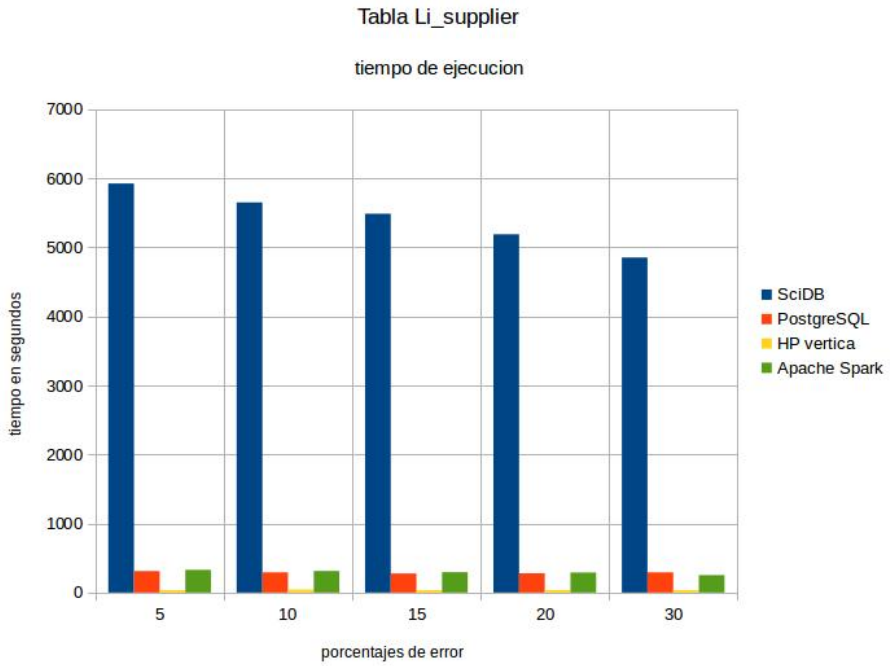


Figure 4.1: Tabla LI_SUPPLIER Tiempo de ejecución.

En el tiempo de ejecución se puede observar que SciDB tarda aproximadamente 160 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, aunque entre mayor sea el porcentaje de error de integridad referencial puede disminuir aproximadamente un 20% del tiempo más alto que registró este sistema. Mientras que PostgreSQL y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica y aunque cambie el porcentaje de error de integridad referencial su tiempo de ejecución no varia significativamente.

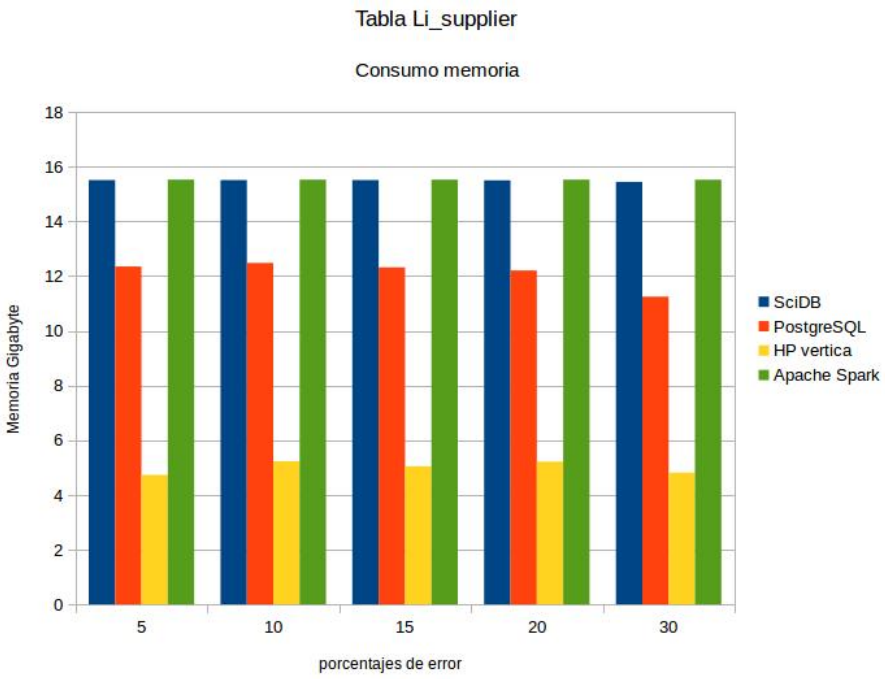


Figure 4.2: Tabla LI_SUPPLIER Consumo de memoria.

En cuanto al consumo de memoria se puede observar que SciDB y Apache Spark consume aproximadamente 3.2 veces más memoria que el sistema que utiliza menos memoria que en este caso es HP Vertica, y sin importar que sea mayor el porcentaje de error de integridad referencial ambos sistemas se mantienen igual. Mientras PostgreSQL que es el segundo sistema que consume más memoria, cuando crece el porcentaje de error de integridad referencial se mantiene practicamente igual, sólo en el último porcentaje disminuye aproximadamente un 10% comparado con el consumo más alto que registró. HP Vertica es el sistema que registra el menor consumo de memoria, este sistema no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución.

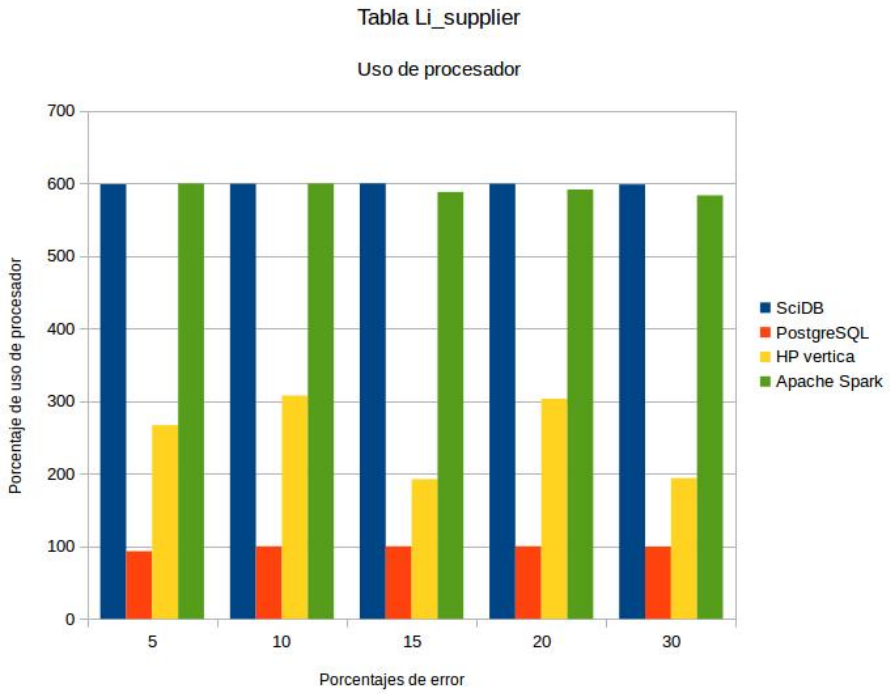


Figure 4.3: Tabla LI_SUPPLIER Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que SciDB y Apache Spark tienen un comportamiento parecido, aunque Apache Spark reduce su uso de procesador cuando crece el porcentaje de error de integridad referencial, estos sistemas utilizan aproximadamente 6 veces más porcentaje de procesador en comparación al sistema que utiliza el menor porcentaje de procesador que en este caso es PostgreSQL. HP vertica es el segundo sistema con menor uso de procesador, este sistema no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. PostgreSQL es el sistema que utiliza el menor porcentaje de procesador, se mantiene fijo no presenta variaciones en su uso de procesador.

### 4.5.2 Tabla tm_ae_max

Los resultados obtenidos al momento de crear la tabla *tm_ae_max* son los que se muestran a continuación.

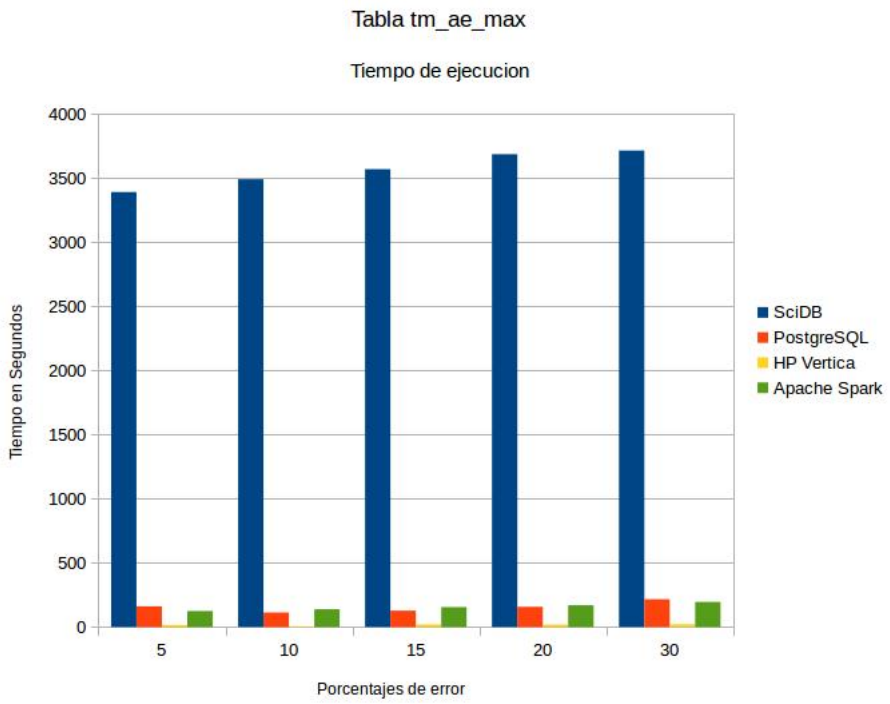


Figure 4.4: Tabla tm_ae_max Tiempo de ejecución.

En el tiempo de ejecución se puede observar que SciDB tarda aproximadamente 210 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, aunque entre mayor sea el porcentaje de error de integridad referencial puede incrementar aproximadamente un 10% del tiempo más bajo que registró este sistema. Mientras que PostgreSQL y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, aunque cambie el porcentaje de error de integridad referencial su tiempo de ejecución no varia significativamente.

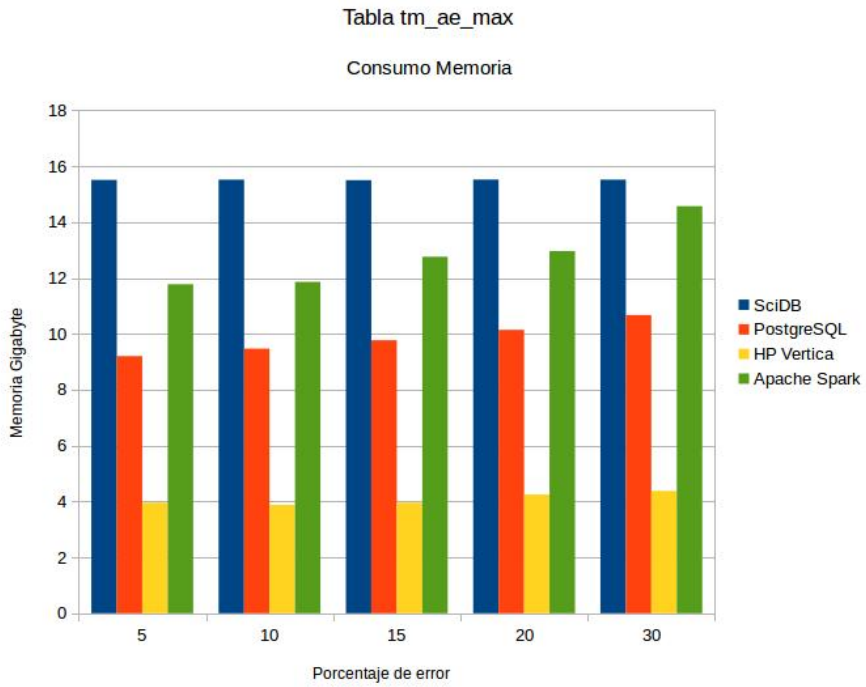


Figure 4.5: Tabla tm_ae_max Consumo de memoria.

En cuanto al consumo de memoria se puede observar que SciDB es el sistema que utiliza más memoria y consume aproximadamente 3.9 veces más memoria que el sistema que utiliza menos memoria que en este caso es HP Vertica, y sin importar que incremente el porcentaje de error de integridad referencial su consumo de memoria se mantiene igual. Apache Spark es el segundo con el consumo de memoria más alto, cuando incrementa el porcentaje de error de integridad referencial su consumo de memoria puede incrementar aproximadamente un 20% respecto al consumo más bajo que registró. Mientras PostgreSQL es el tercer sistema con el consumo más alto de memoria, cuando crece el porcentaje de error de integridad referencial, el porcentaje de uso de memoria puede incrementar aproximadamente un 15% respecto al consumo más bajo que registró este sistema. HP Vertica es el sistema que registra el menor consumo de memoria, aunque cambie el porcentaje de error de integridad referencial su consumo de memoria no varia significativamente.

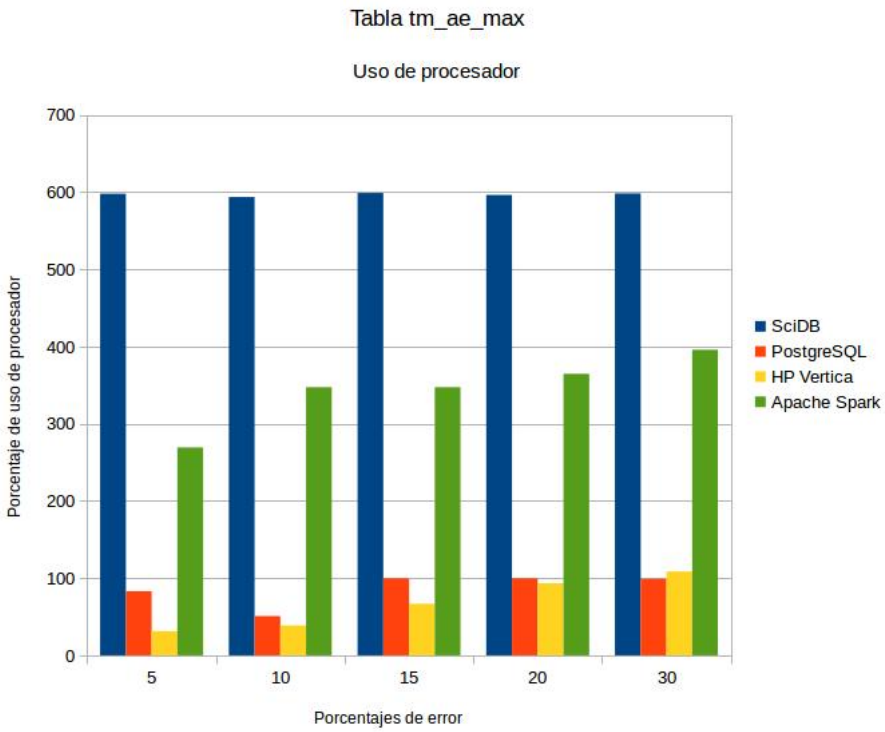


Figure 4.6: Tabla tm_ae_max Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que SciDB tiene el uso de procesador más alto, este porcentaje no cambia cuando incrementa el porcentaje de error de integridad referencial, este sistema utiliza aproximadamente 19 veces mas porcentaje de procesador que el sistema que utiliza menos que en este caso es HP Vertica. El segundo sistema que utiliza el mayor porcentaje de procesador es Apache spark, cuando se incrementa el porcentaje de error de integridad referencial, su porcentaje de uso de procesador puede incrementar aproximadamente un 45% comparado con el porcentaje más bajo que registró. El tercer sistema con el mayor porcentaje de uso de procesador utilizado es PostgreSQL, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. El sistema con el menor uso de procesador es HP Vertica en este sistema cuando incrementa el porcentaje de error de integridad referencial, su porcentaje de uso de procesador puede incrementar aproximadamente 3.4 veces comparado con el porcentaje más bajo que registró.

### 4.5.3 Tabla tm_ae_min

Los resultados obtenidos al momento de crear la tabla tm_ae_min son los que se muestran a continuación.

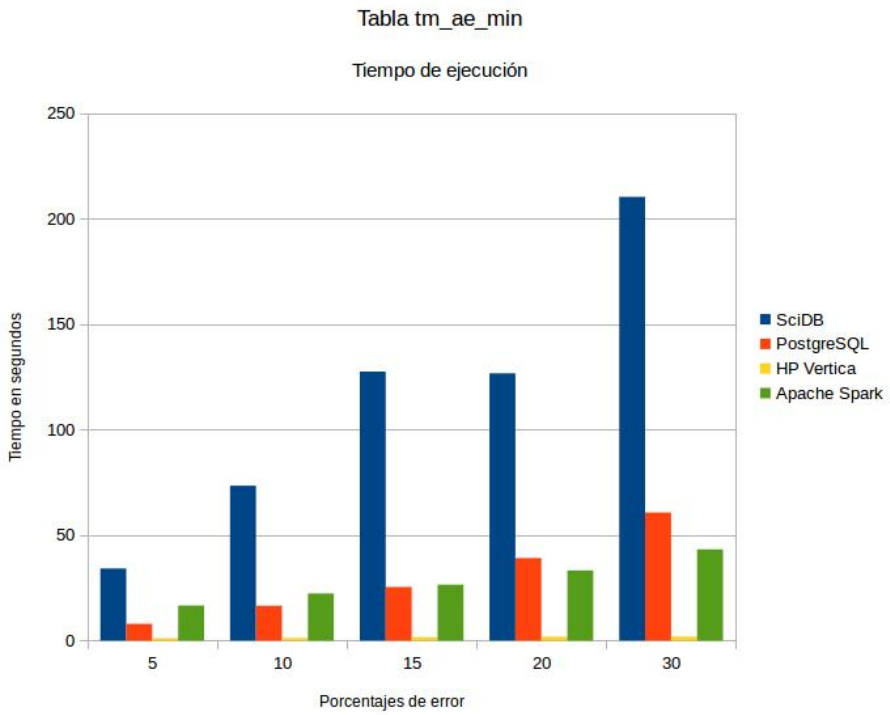


Figure 4.7: Tabla tm_ae_min Tiempo de ejecución.

En el tiempo de ejecución se puede observar que SciDB puede tardar hasta aproximadamente 115 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 6.1 veces entre el tiempo más bajo y más alto que registró. Mientras que PostgreSQL y Apache Spark son muy parecidos en comportamiento, en ambos sistemas el tiempo de ejecucion incrementa con el incremento del porcentaje de error de integridad referencial, en este caso el incremento es 7.6 veces para PostgreSQL y 2.6 veces para Apache spark, esto respecto a a los tiempos más bajos y más altos que registraron. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial su tiempo de ejecución no varia significativamente.

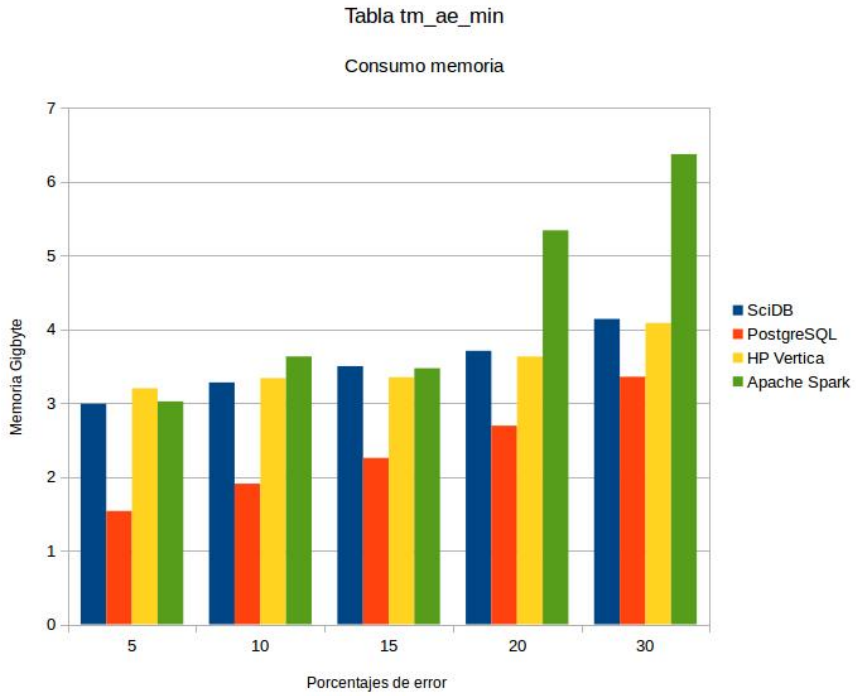


Figure 4.8: Tabla tm_ae_min Consumo de memoria.

En cuanto al consumo de memoria se puede observar que Apache Spark es el sistema que utiliza más memoria y consume aproximadamente 1.9 veces más memoria que el sistema que utiliza menos que en este caso es PostgreSQL, aunque entre mayor sea el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente 2.1 veces respecto al consumo más bajo que registró. SciDB es el segundo sistema con el consumo de memoria más grande, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente un 40% respecto al consumo más bajo que registró. Mientras Apache Spark es el tercer sistema con el consumo de memoria más grande, cuando incrementa el porcentaje de error de integridad referencial, el consumo de memoria puede incrementar aproximadamente un 25% respecto al consumo más bajo que registró. PostgreSQL es el sistema que registra el menor consumo de memoria, cuando incrementa el porcentaje de error de integridad referencial, el porcentaje de consumo de memoria puede incrementar aproximadamente 2.2 veces respecto al consumo más bajo que registró este sistema.

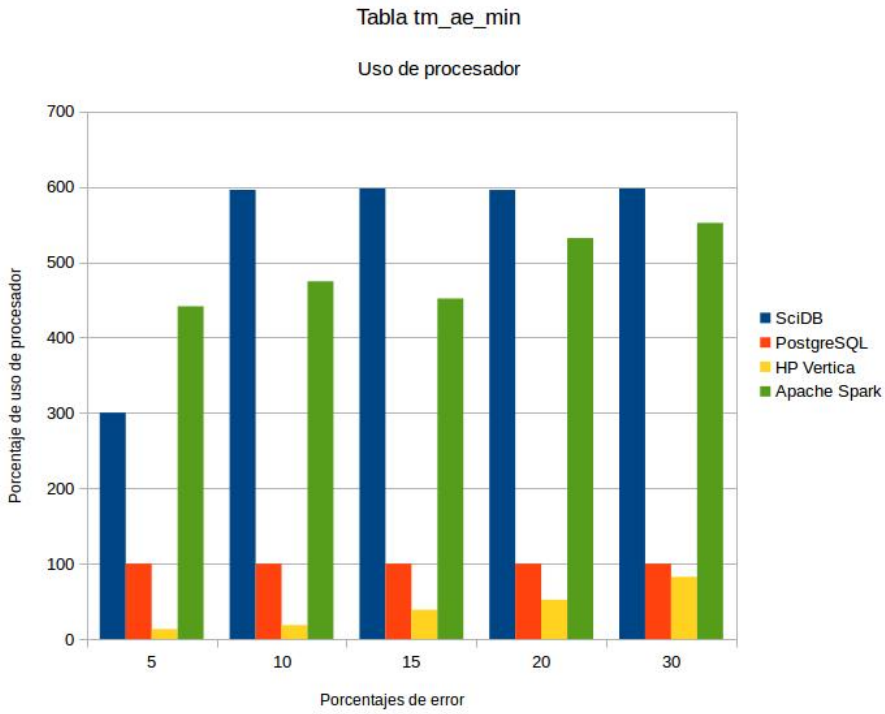


Figure 4.9: Tabla tm_ae_min Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que SciDB tiene el uso de procesador más alto, el cual incrementa al doble en comparación entre la primera y las demás ejecuciones, después de esto, su porcentaje de uso de procesador no cambia cuando incrementa el porcentaje de error de integridad referencial, este sistema utiliza aproximadamente 33 veces mas porcentaje de procesador que el sistema que utiliza menos que en este caso es HP Vertica. El segundo sistema que utiliza el mayor porcentaje de procesador es Apache spark, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. El tercer sistema con el mayor porcentaje de procesador utilizado es PostgreSQL, el cual no cambia cuando incrementa el porcentaje de error de integridad referencial. El sistema con el menor uso de procesador es HP Vertica, en este sistema cuando incrementa el porcentaje de error de integridad referencial su uso de procesador puede incrementar aproximadamente 6.2 veces comparado con el porcentaje más bajo que registró.

### 4.5.4 Tabla tm_fw

Los resultados obtenidos al momento de crear la tabla tm_fw son los que se muestran a continuación.

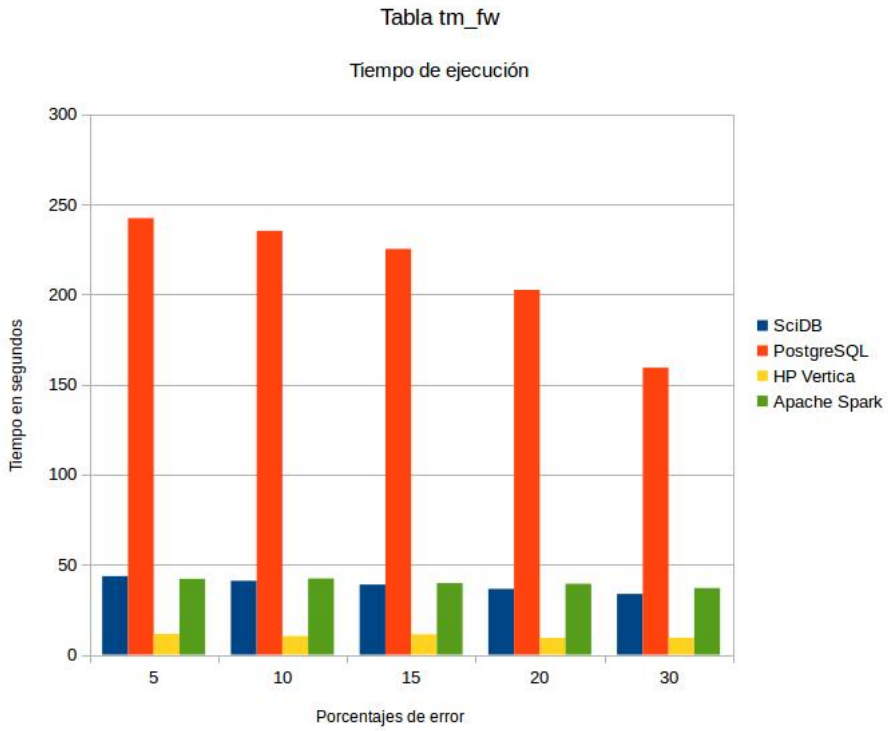


Figure 4.10: Tabla tm_fw Tiempo de ejecución.

En el tiempo de ejecución se puede observar que PostgreSQL puede tardar hasta aproximadamente 22 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 52% entre el tiempo más bajo y más alto que registró. Mientras que PostgreSQL y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial su tiempo de ejecución no varia significativamente.

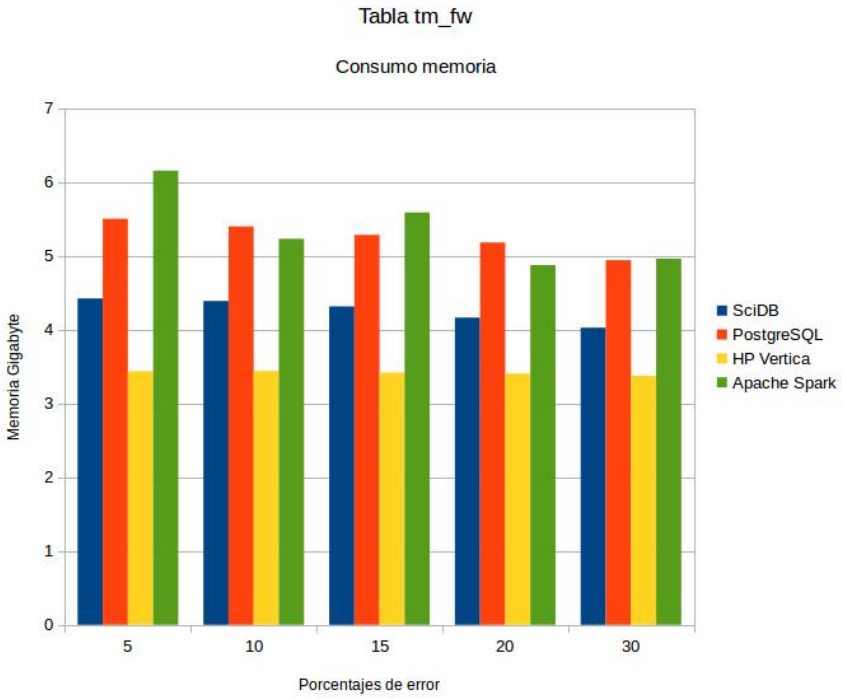


Figure 4.11: Tabla tm_fw Consumo de memoria.

En cuanto al consumo de memoria se puede observar que Apache Spark es el sistema que utiliza más memoria y consume aproximadamente un 75% más memoria que el sistema que utiliza menos que en este caso es HP Vertica, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. PostgreSQL es el segundo sistema con el consumo más alto de memoria, cuando incrementa el porcentaje de error de integridad referencial, su uso de memoria puede incrementar aproximadamente un 10% respecto al consumo más bajo que registró. Mientras SciDB es el tercer sistema con el consumo más alto de memoria, cuando incrementa el porcentaje de error de integridad referencial, su uso de memoria puede decrementar aproximadamente un 10% respecto al consumo más alto que registró. HP Vertica es el sistema que registra el menor consumo de memoria, y sin importar que incremente el porcentaje de error de integridad referencial su consumo de memoria se mantiene igual.

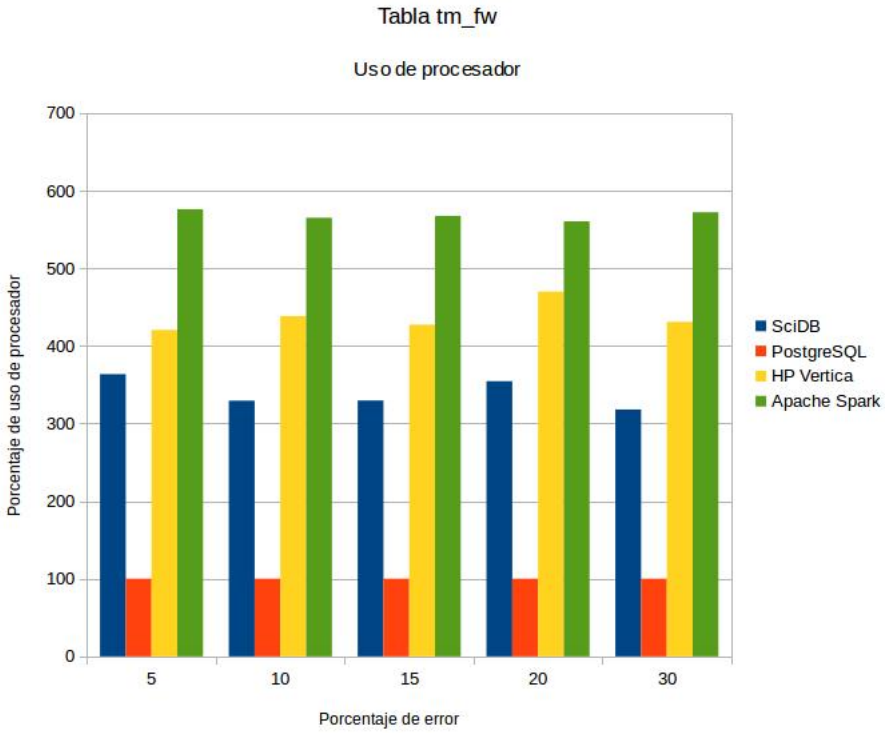


Figure 4.12: Tabla tm_fw Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que Apache Spark tiene el uso de procesador más alto, este porcentaje no cambia cuando incrementa el porcentaje de error de integridad referencial, este sistema utiliza aproximadamente 5.8 veces mas porcentaje de procesador que el sistema que utiliza menos que en este caso es PostgreSQL. El segundo sistema que utiliza el mayor porcentaje de procesador es HP Vertica, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. El tercer sistema con el mayor porcentaje de procesador utilizado es SciDB, este sistema tampoco tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. El sistema con el menor uso de procesador es PostgreSQL, este sistema se mantiene igual sin importar cual sea el porcentaje de error de integridad referencial.

### 4.5.5 Conteo por llave primaria

Los resultados obtenidos al momento de ejecutar la función para obtener el Conteo por llave primaria son los que se muestran a continuación.

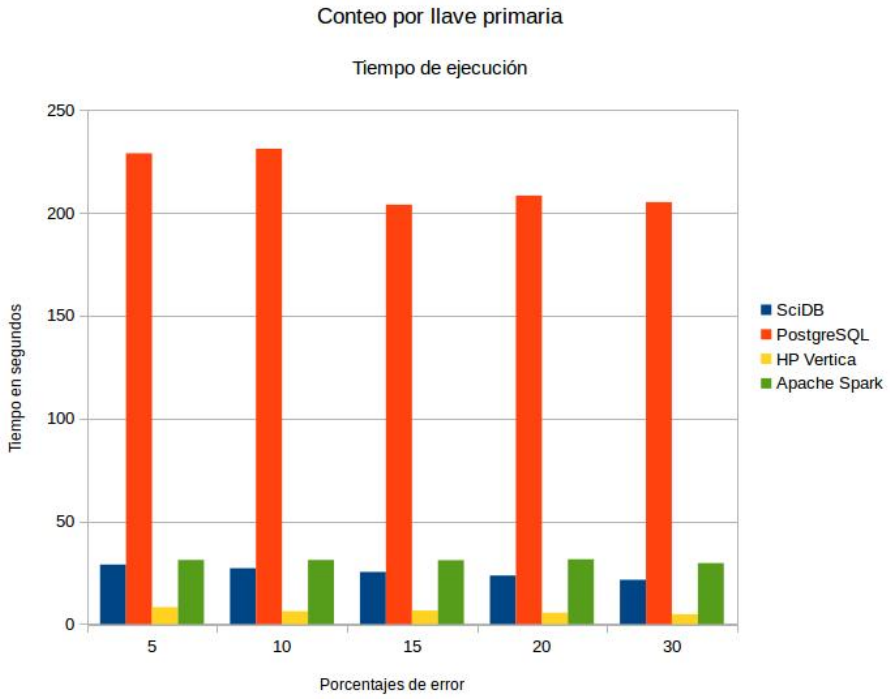


Figure 4.13: Conteo por llave primaria Tiempo de ejecución con tablas auxiliares.

En el tiempo de ejecución se puede observar que PostgreSQL es el sistema con el tiempo de ejecución más alto y tarda aproximadamente 20 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede disminuir aproximadamente un 20% en comparación con el tiempo más alto que registró. Mientras que SciDB y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede disminuir aproximadamente un 20% en comparación con el tiempo de ejecución más alto que registró.

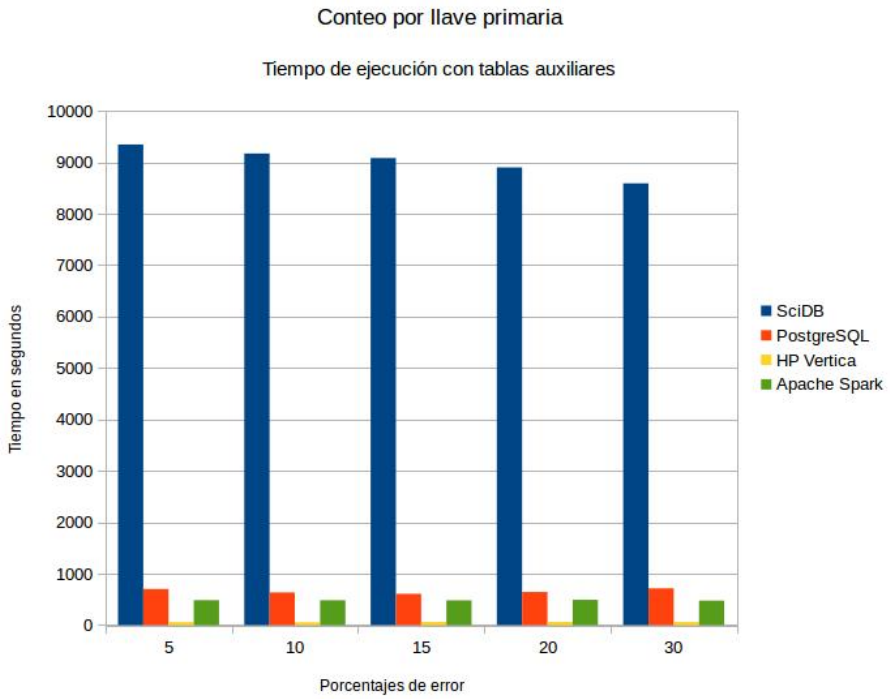


Figure 4.14: Conteo por llave primaria Tiempo de ejecución.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es SciDB, seguido por PostgreSQL, después Apache Spark y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es HP Vertica.

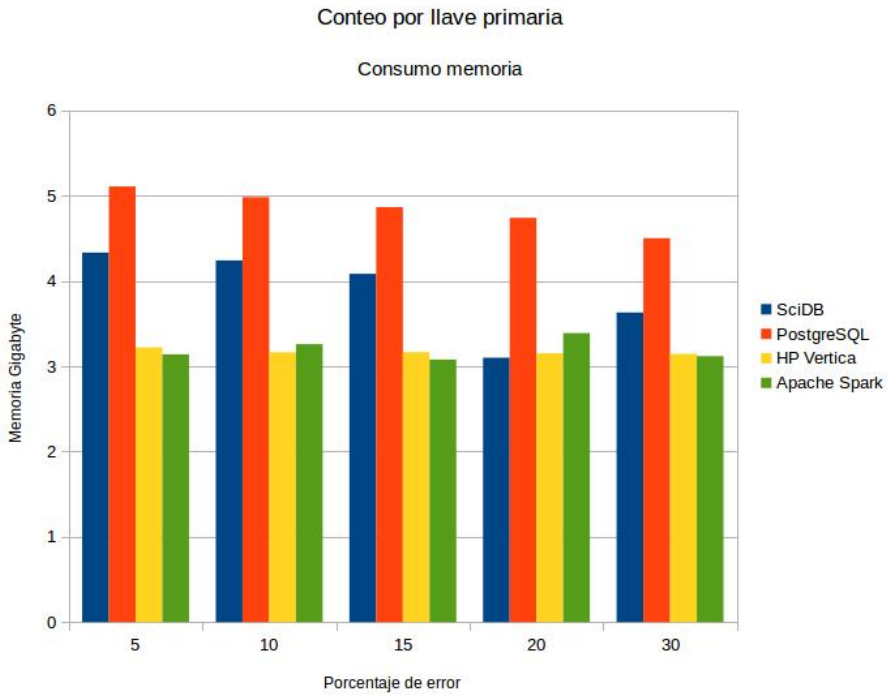


Figure 4.15: Conteo por llave primaria Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema con el consumo más alto de memoria y este sistema consume aproximadamente 60% más memoria que los sistemas que consumen menos que en este caso son HP Vertica y Apache Spark, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede disminuir aproximadamente un 15% comparado con el consumo más alto que registró. Mientras SciDB que es el segundo sistema con el consumo más alto de memoria, cuando incrementa el porcentaje de error de integridad referencial, su uso de memoria puede disminuir aproximadamente un 15% comparado con el consumo más alto que registró. Apache Spark y HP Vertica son los sistemas que registran los menores consumos de memoria, son muy parecidos, en cuanto a tiempo y comportamiento.

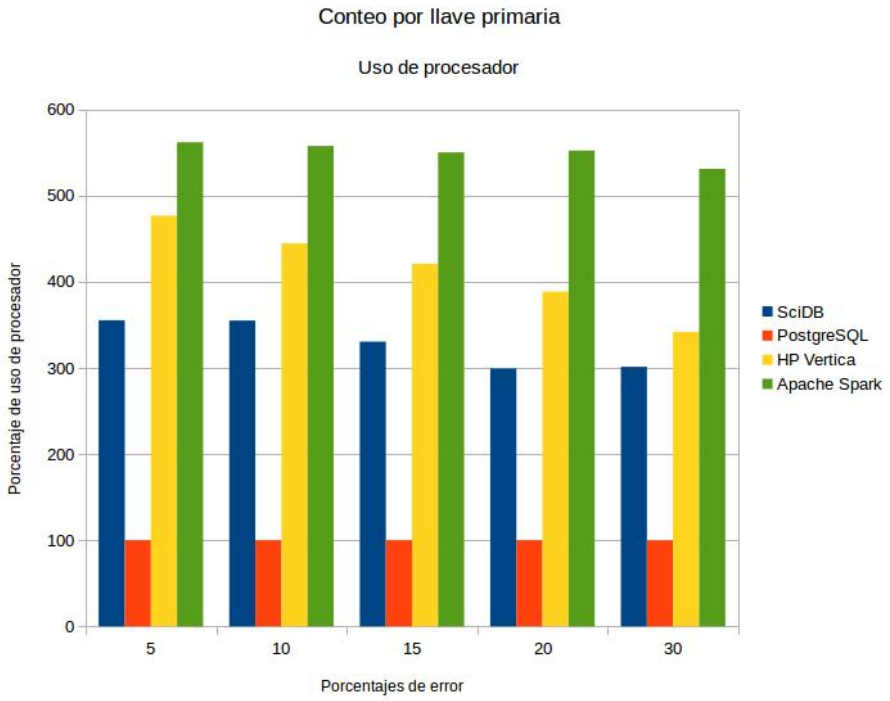


Figure 4.16: Conteo por llave primaria Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que PostgreSQL tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial, es aproximadamente 5.5 veces menor que el sistema que utiliza el mayor porcentaje de procesador que en este caso es Apache Spark. SciDB es el segundo sistema con menor uso de procesador, tiene un decremento del 10% entre el porcentaje de procesador más alto y el más bajo que registró. En cuanto a HP vertica, cuando incrementa el porcentaje de error de integridad referencial, su uso de procesador decrementa aproximadamente un 40% en comparación al porcentaje más grande que registró. Mientras que Apache Spark es el sistema con el mayor uso de procesador el cual no presenta mucha variación en cuanto al porcentaje de error de integridad referencial.

### 4.5.6 Conteo total por llave primaria

Los resultados obtenidos al momento de ejecutar la función para obtener el Conteo total por llave primaria son los siguientes:

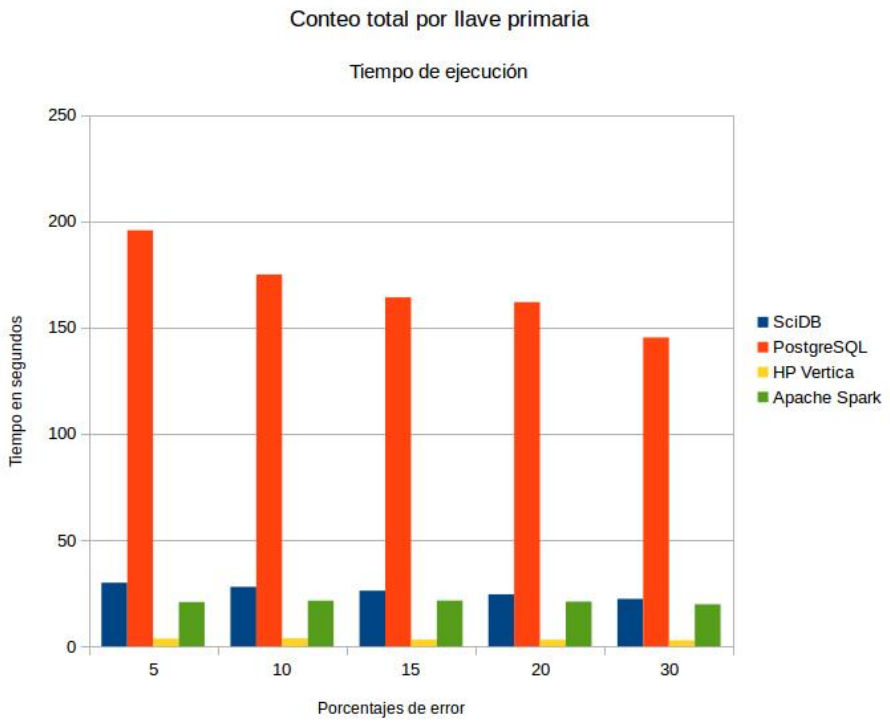


Figure 4.17: Conteo total por llave primaria Tiempo de ejecución.

En el tiempo de ejecución se puede observar que PostgreSQL tiene el tiempo de ejecución más alto y tarda aproximadamente 35 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 30% en comparación al tiempo más alto que registró. Mientras que SciDB y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 20% en comparación al tiempo de ejecución más alto que registró.

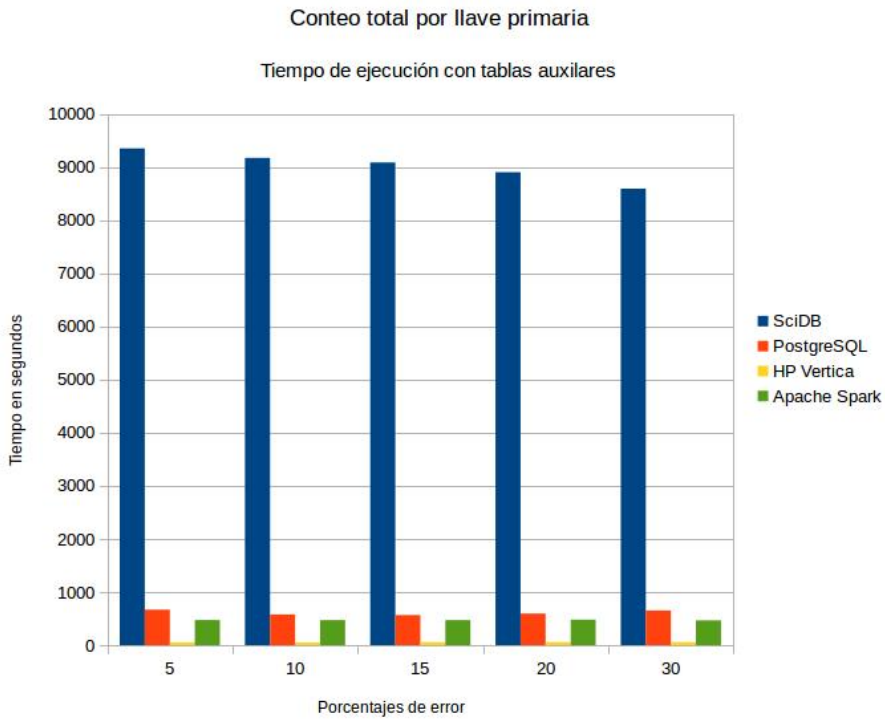


Figure 4.18: Conteo total por llave primaria Tiempo de ejecución con tablas auxiliares.

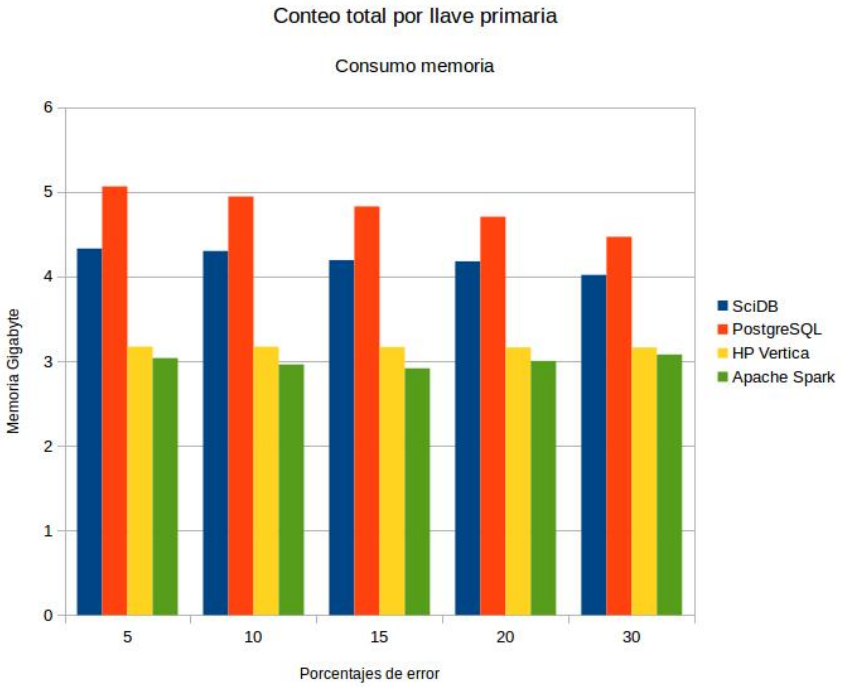


Figure 4.19: Conteo total por llave primaria Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema con el consumo de memoria más alto y consume aproximadamente 60% más memoria que los sistemas que tienen el consumo de memoria más bajo, que en este caso son HP Vertica y Apache Spark, cuando incrementa el porcentaje de error de integridad referencial, su uso de memoria puede decrementar aproximadamente un 10% comparado con el consumo más alto que registró. Mientras SciDB es el segundo sistema con el consumo de memoria más alto, cuando incrementa el porcentaje de error de integridad referencial su consumo de memoria puede decrementar aproximadamente un 5% comparado con el consumo más alto que registró. Apache Spark y HP Vertica son los sistemas que registran los menores consumos de memoria, son muy parecidos, en cuanto a tiempo y comportamiento.

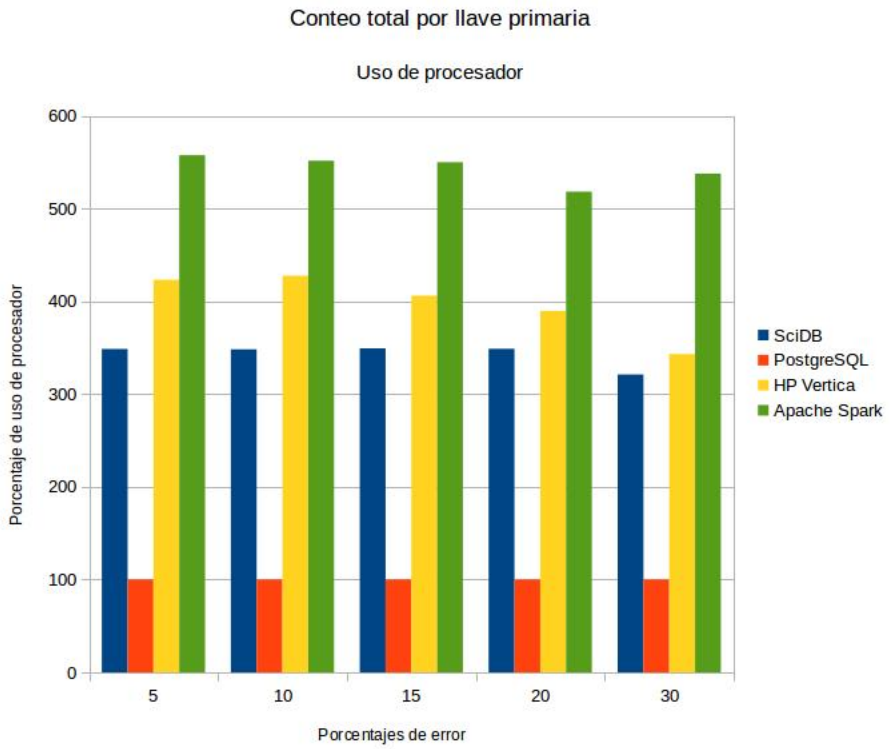


Figure 4.20: Conteo total por llave primaria Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que PostgreSQL es el sistema con el menor porcentaje de uso de procesador además tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial, es aproximadamente 5.5 veces menor que el sistema que utiliza el mayor porcentaje de procesador que es Apache Spark. SciB es el segundo sistema con menor uso de procesador, tiene un decremento del 10% entre el porcentaje de procesador más alto y el más bajo que registró. HP vertica, cuando incrementa el porcentaje de error de integridad referencial, su uso de procesador decrementa aproximadamente un 25%. Mientras que Apache Spark es el sistema con el mayor uso de procesador, cuando incrementa el porcentaje de error de integridad referencial, su comportamiento no cambia.

### 4.5.7 Conteo por columna especificada

Los resultados obtenidos al momento de ejecutar la función para obtener el Conteo por columna especificada son los siguientes:

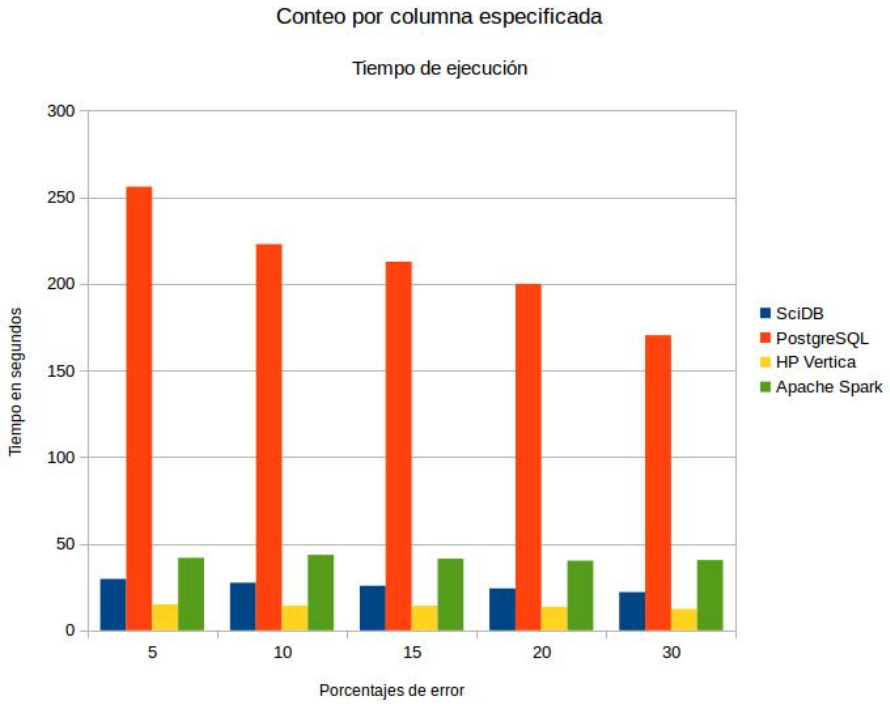


Figure 4.21: Conteo por columna especificada Tiempo de ejecución.

En el tiempo de ejecución se puede observar que PostgreSQL tiene el tiempo de ejecución más alto y puede tardar aproximadamente 11 veces más tiempo que el sistema con el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 45% del tiempo más alto que registró. Mientras que Apache Spark es el segundo sistema con el mayor tiempo de ejecución, cuando incrementa el porcentaje de error de integridad referencial, su comportamiento no cambia. En cuanto a los sistemas que tiene el menor tiempo de ejecución son SciDB y HP Vertica, aunque HP Vertica tiene un menor tiempo de ejecución comparado con SciDB, esto en aproximadamente un 30%.

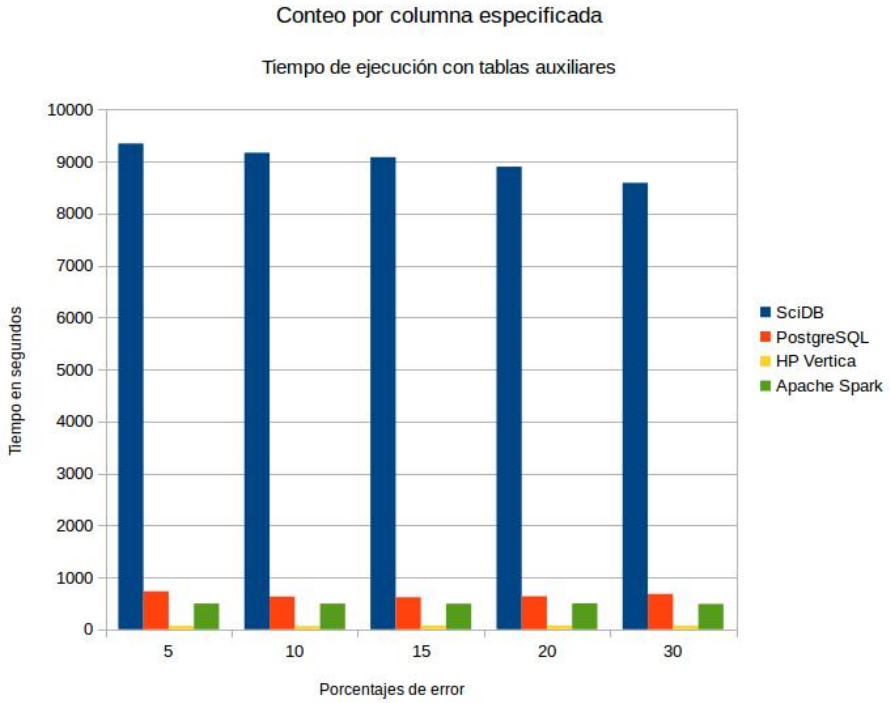


Figure 4.22: Conteo por columna especificada Tiempo de ejecución con tablas auxiliares.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es SciDB, seguido por PostgreSQL, después Apache Spark y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es HP Vertica.

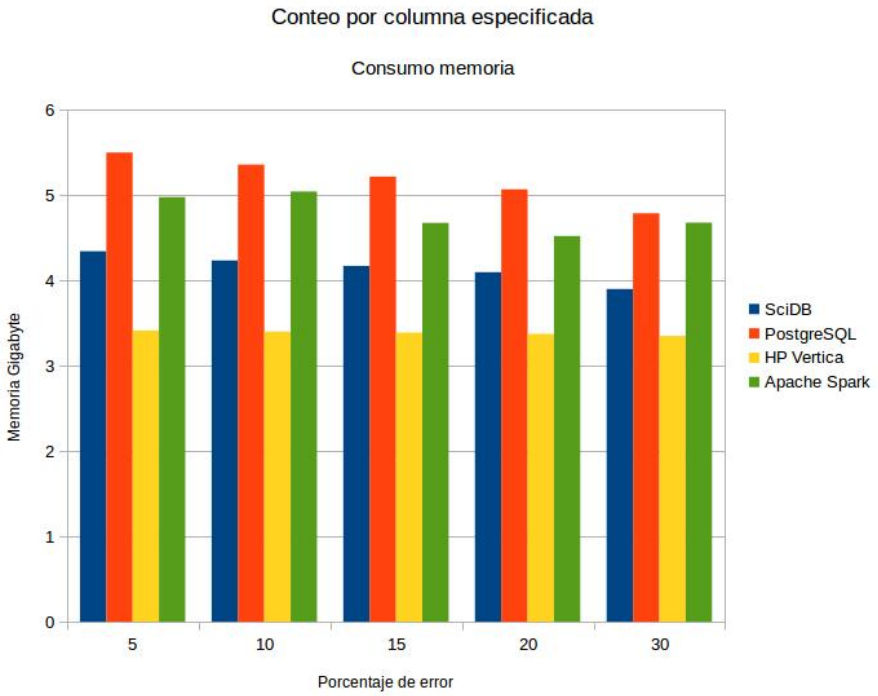


Figure 4.23: Conteo por columna especificada Consumo de memoria.

En cuanto al uso de memoria se puede observar que PostgreSQL es el sistema que utiliza la mayor cantidad de memoria y consume aproximadamente un 60% más memoria que el sistema que utiliza menos que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su uso de memoria puede decrementar aproximadamente un 15% comparado con el uso de memoria más alto que registró. Mientras Apache Spark es el segundo sistema con el uso de memoria más alto, no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. El tercer sistema con el uso de memoria más alto es SciDB, cuando incrementa el porcentaje de error de integridad referencial, su uso de memoria puede decrementar aproximadamente un 15% comparado con el uso más alto que registró. HP Vertica es el sistema que registra el menor uso de memoria, y sin importar que incremente el porcentaje de error de integridad referencial su uso de memoria se mantiene igual.

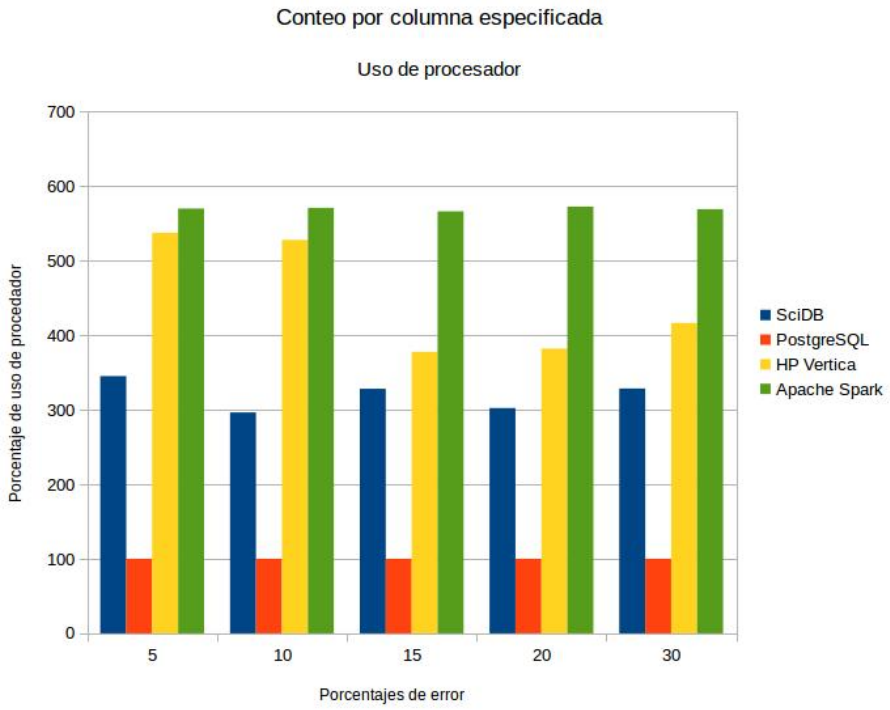


Figure 4.24: Conteo por columna especificada Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que PostgreSQL tiene el uso de procesador más bajo, además tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial, es aproximadamente 5.8 veces menor que el sistema que utiliza el mayor porcentaje de procesador que en este caso es Apache Spark. Hp Vertica es el segundo sistema con menor uso de procesador, no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. El tercer sistema con el uso de procesador más bajo es SciDB tampoco tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras que Apache Spark tiene el uso de procesador más alto, este porcentaje no cambia cuando incrementa el porcentaje de error de integridad referencial.

### 4.5.8 Conteo total por columna especificada

Los resultados obtenidos al momento de ejecutar la función para obtener el Conteo total por columna especificada son los siguientes:

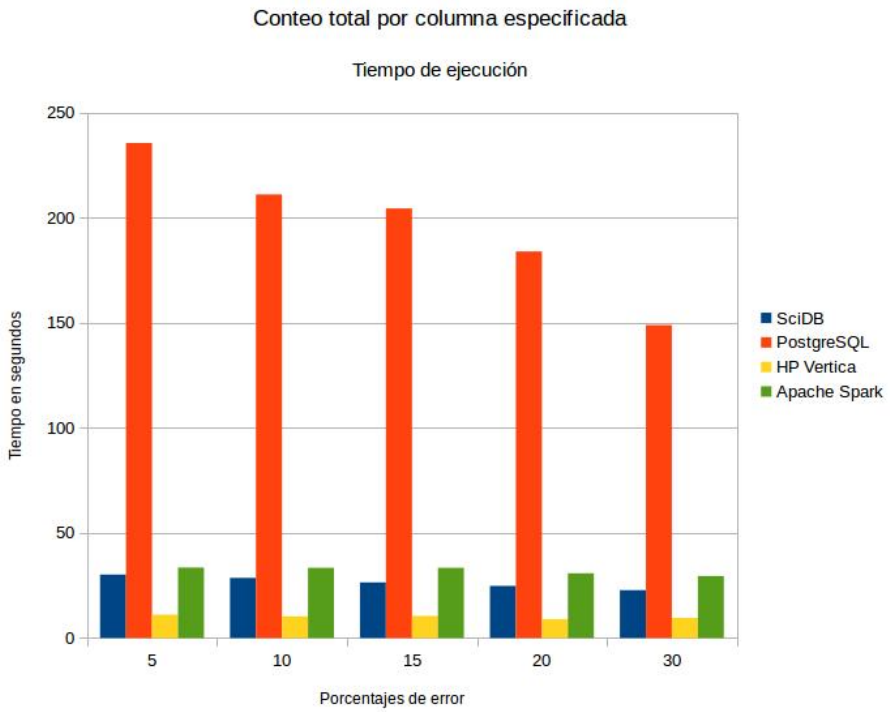


Figure 4.25: Conteo total por columna especificada Tiempo de ejecución.

En el tiempo de ejecución se puede observar que PostgreSQL es el sistema con el mayor tiempo de ejecución y puede tardar aproximadamente 12 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 60% entre el tiempo más bajo y más alto que registró. Mientras que SciDB y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 20% entre el tiempo de ejecución más bajo y más alto que registró.

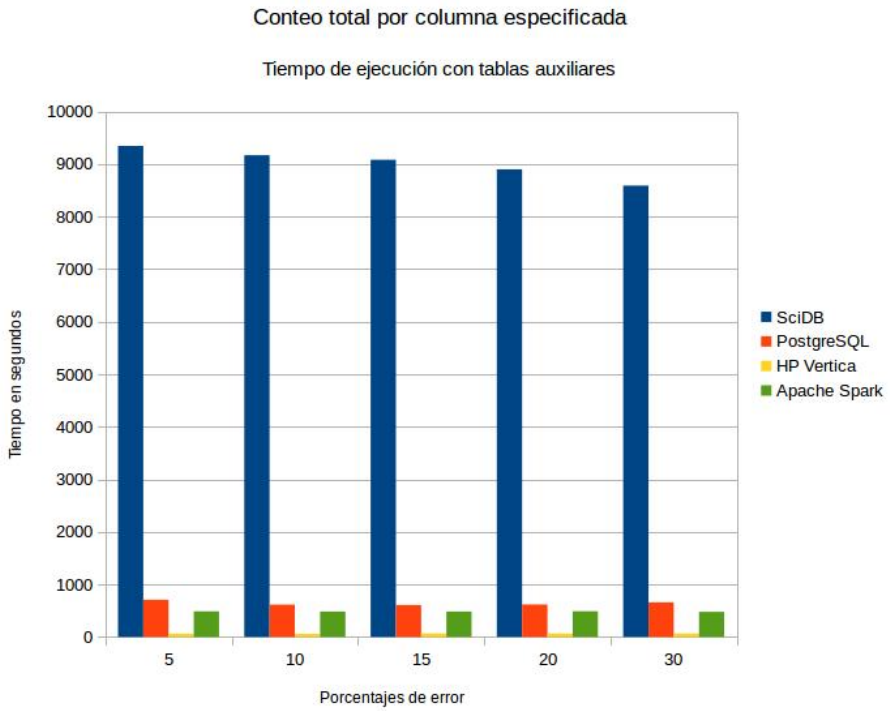


Figure 4.26: Conteo total por columna especificada Tiempo de ejecución con tablas auxiliares.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es SciDB, seguido por PostgreSQL, después Apache Spark y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es HP Vertica.

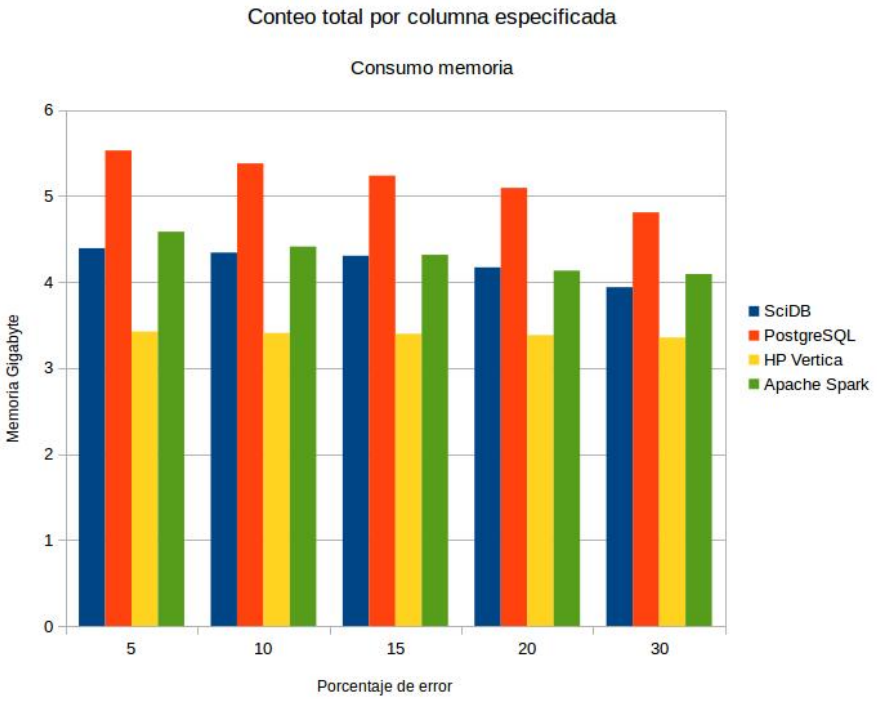


Figure 4.27: Conteo total por columna especificada Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema que utiliza más memoria y consume aproximadamente 55% más memoria que el sistema que utiliza menos que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 15% comparado con el consumo más alto que registró. Mientras Apache Spark y SciDB son muy parecidos en cuanto a consumo de memoria y comportamiento, decrementando conforme incrementa el porcentaje de error de integridad referencial, esto aproximadamente un 15%. HP Vertica es el sistema que registra el menor consumo de memoria, manteniéndose igual sin importar el porcentaje de error de integridad referencial.

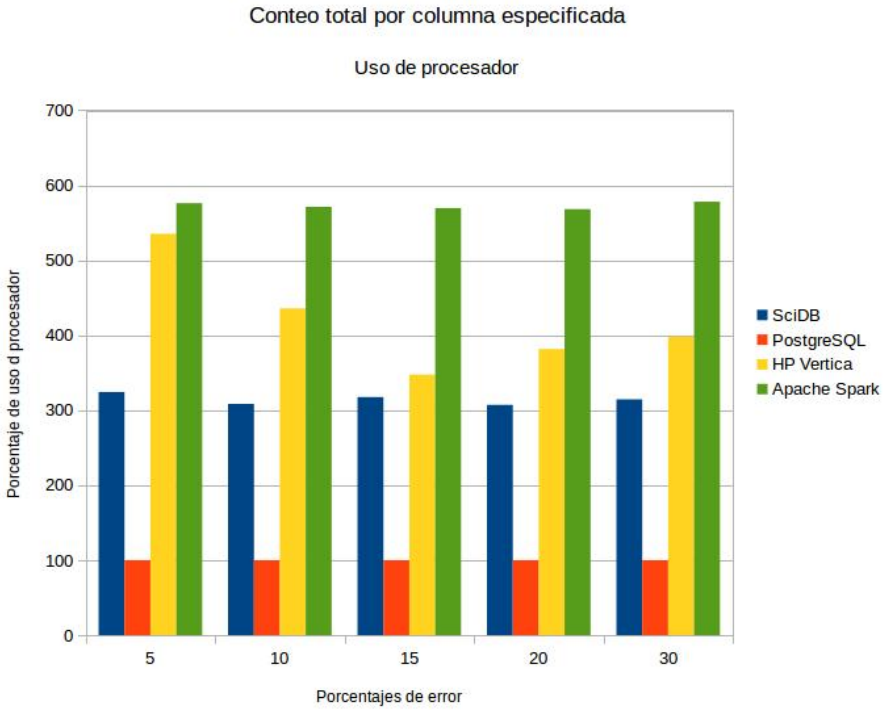


Figure 4.28: Conteo total por columna especificada Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que PostgreSQL tiene el uso de procesador más bajo, además tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial, es aproximadamente 5.8 veces menor que el sistema que utiliza el mayor porcentaje de procesador que en este caso es Apache Spark. SciB es el segundo sistema con menor uso de procesador, no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. HP vertica es el tercer sistema con el menor uso de procesador, no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras que Apache Spark tiene el uso de procesador más alto, este porcentaje no cambia cuando incrementa el porcentaje de error de integridad referencial.

### 4.5.9 Suma por columna especificada

Los resultados obtenidos al momento de ejecutar la función para obtener la Suma por columna especificada son los siguientes:

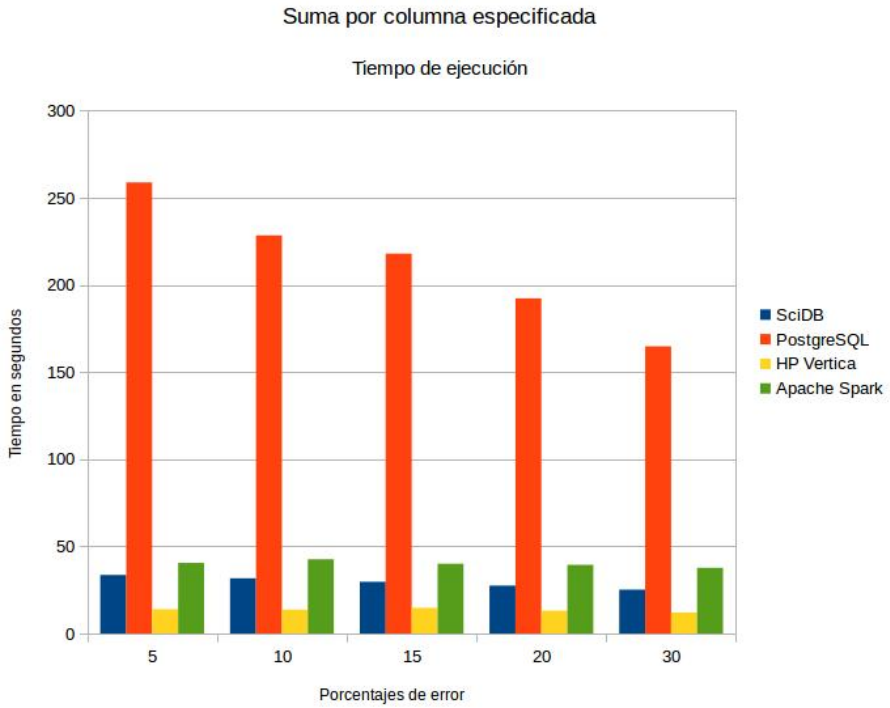


Figure 4.29: Suma por columna especificada Tiempo de ejecución.

En el tiempo de ejecución se puede observar que PostgreSQL es el sistema con el mayor tiempo de ejecución y puede tardar aproximadamente 11 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 30% entre el tiempo más bajo y más alto que registró. Mientras que SciDB y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 20% entre el tiempo de ejecución más bajo y más alto que registró.

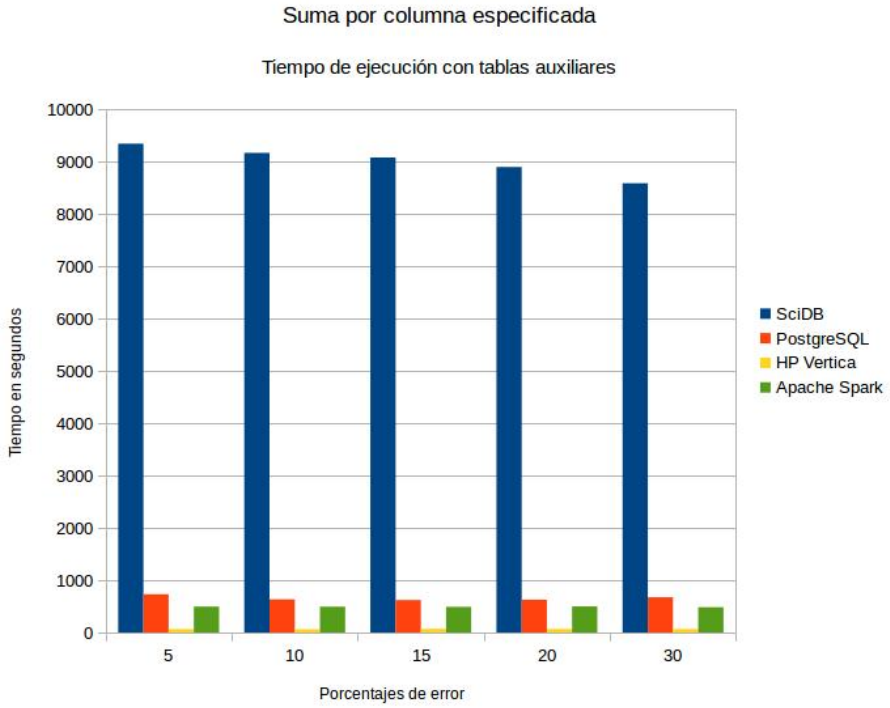


Figure 4.30: Suma por columna especificada Tiempo de ejecución con tablas auxiliares.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es SciDB, seguido por PostgreSQL, después Apache Spark y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es HP Vertica.

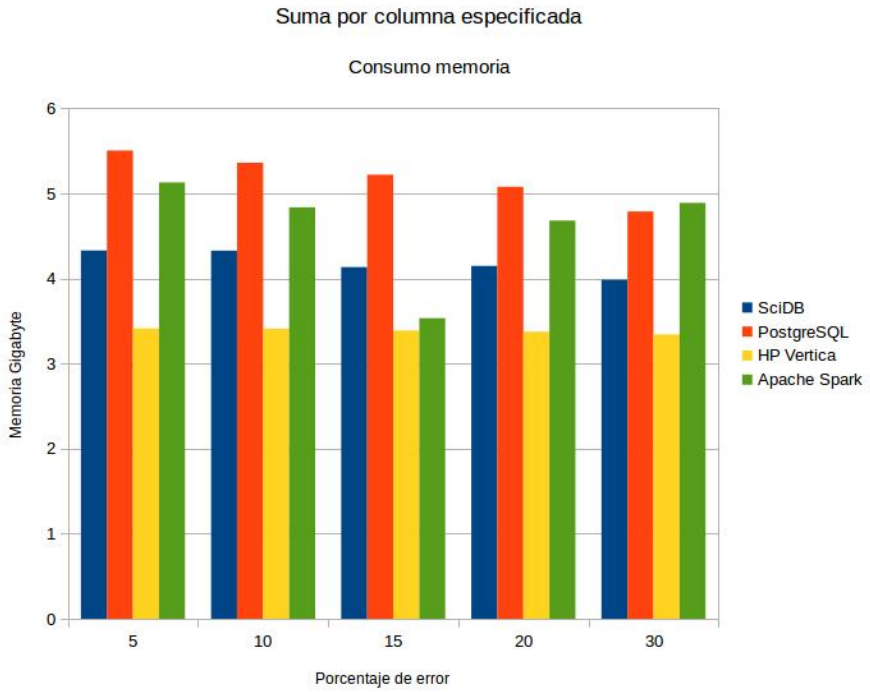


Figure 4.31: Suma por columna especificada Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema que utiliza más memoria y consume aproximadamente 60% más memoria que el sistema que utiliza menos que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 15% comparado con el consumo más alto que registró. Apache Spark es el segundo sistema con el consumo de memoria más alto, no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras SciDB es el tercer sistema con el consumo de memoria más alto, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 15% comparado con el consumo más alto que registró. HP Vertica es el sistema que registra el menor consumo de memoria, manteniéndose igual sin importar el porcentaje de error de integridad referencial.

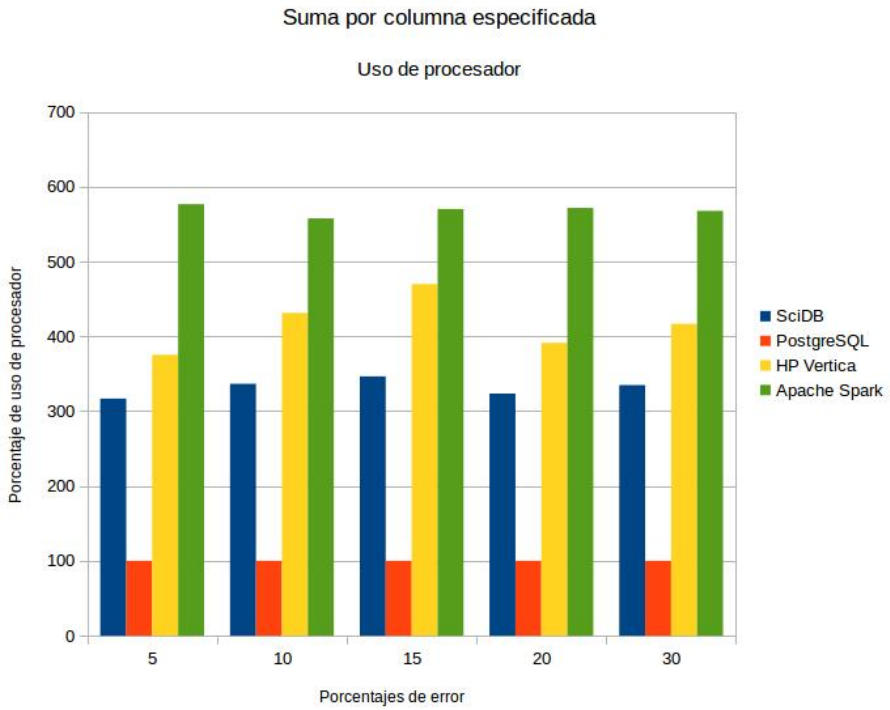


Figure 4.32: Suma por columna especificada Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que PostgreSQL tiene el uso de procesador más bajo, además tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial, es aproximadamente 5.8 veces menor que el sistema que utiliza el mayor porcentaje de procesador que en este caso es Apache Spark. SciB es el segundo sistema con menor uso de procesador, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. HP vertica es el tercer sistema con menor uso de procesador, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras que Apache Spark tiene el uso de procesador más alto, este porcentaje no cambia cuando incrementa el porcentaje de error de integridad referencial.

### 4.5.10 Suma total por columna especificada

Los resultados obtenidos al momento de ejecutar la función para obtener la Suma total por columna especificada son los siguientes:

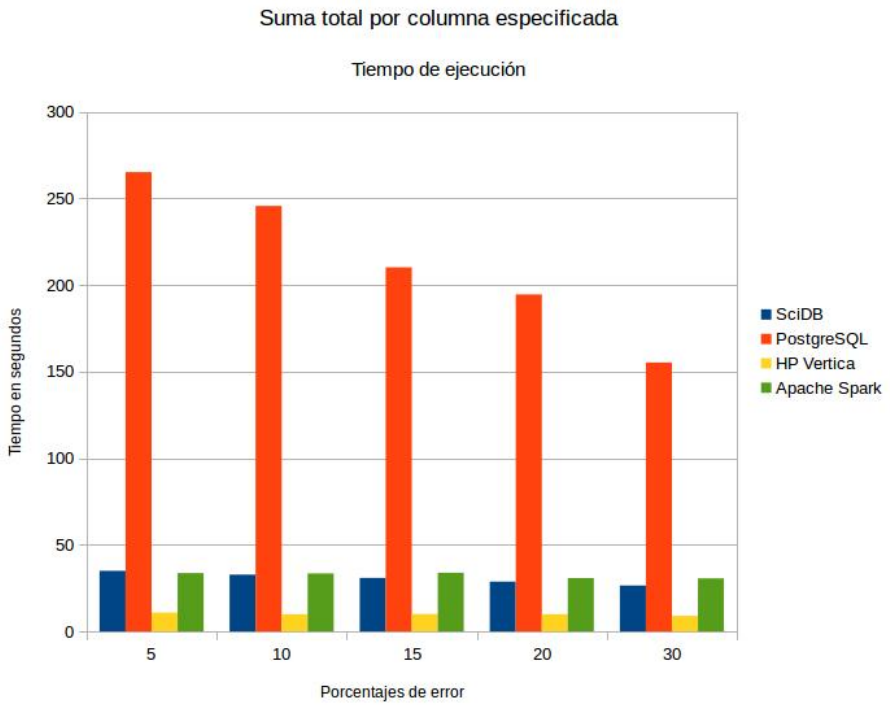


Figure 4.33: Suma total por columna especificada Tiempo de ejecución.

En el tiempo de ejecución se puede observar que PostgreSQL es el sistema con el mayor tiempo de ejecucióon y puede tardar aproximadamente 13 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 70% entre el tiempo más bajo y más alto que registró. Mientras que SciDB y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 15% entre el tiempo de ejecución más bajo y más alto que registró.

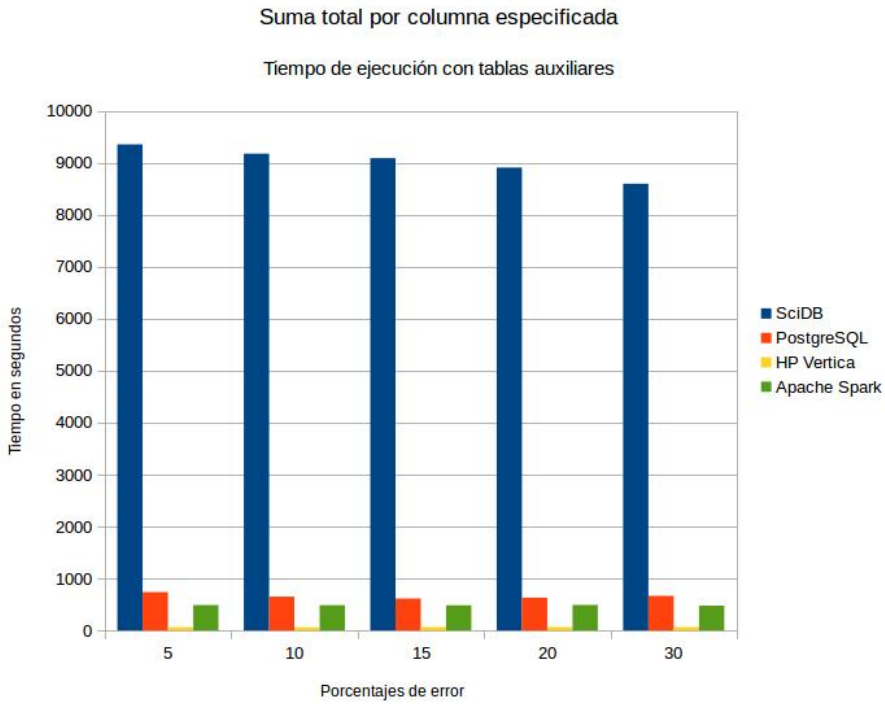


Figure 4.34: Suma total por columna especificada Tiempo de ejecución con tablas auxiliares.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es SciDB, seguido por PostgreSQL, después Apache Spark y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es HP Vertica.

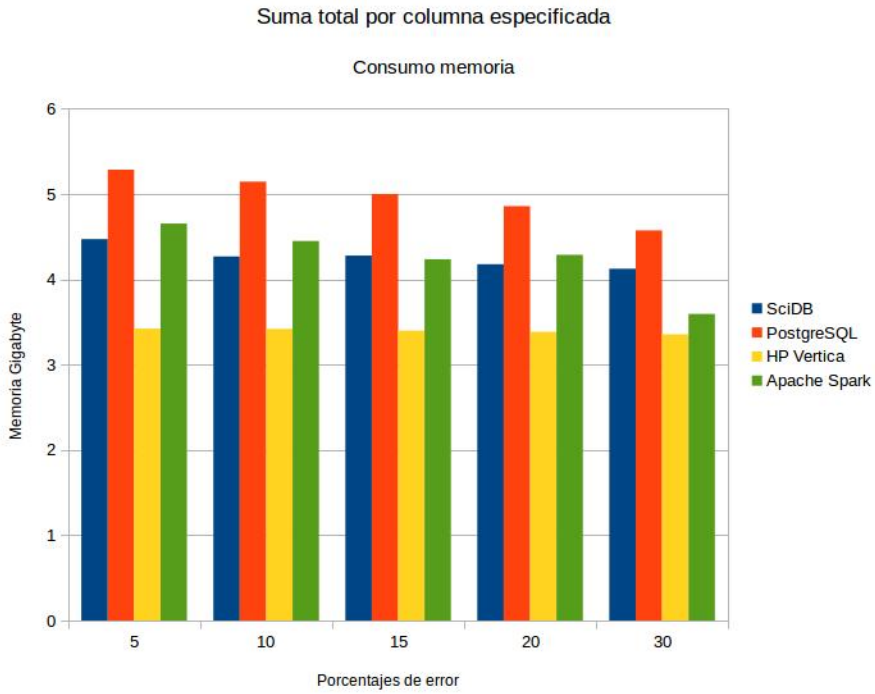


Figure 4.35: Suma total por columna especificada Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema que utiliza más memoria y consume aproximadamente 50% más memoria que el sistema que utiliza menos que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 15% comparado con el consumo más alto que registró. Apache Spark es el segundo sistema con el consumo de memoria más alto, no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras SciDB es el tercer sistema con el consumo de memoria más alto, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 10% comparado con el consumo más alto que registró. HP Vertica es el sistema que registra el menor consumo de memoria, manteniéndose igual sin importar el porcentaje de error de integridad referencial.

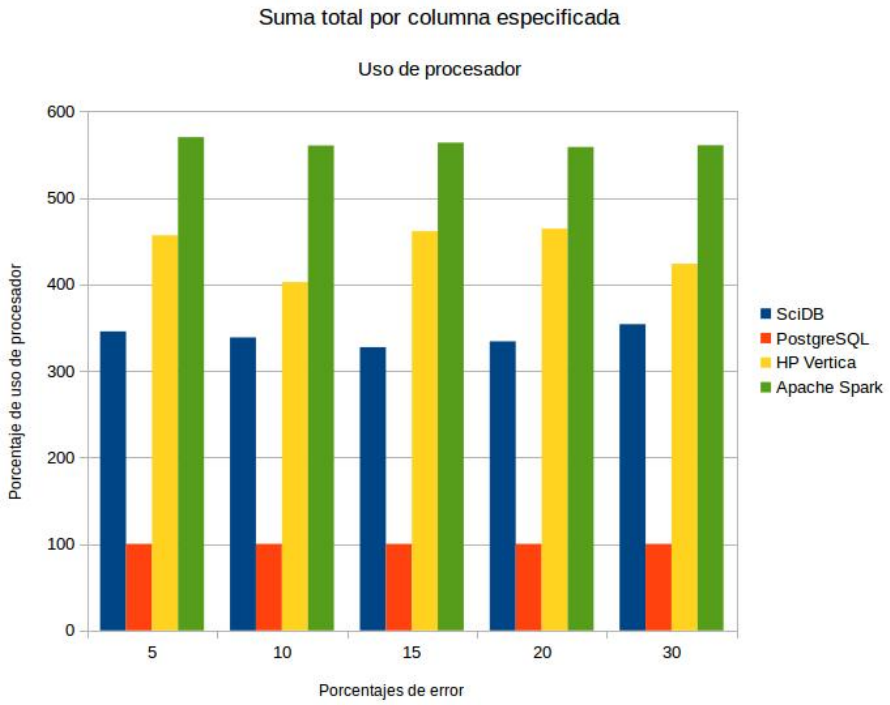


Figure 4.36: Suma total por columna especificada Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que PostgreSQL tiene el uso de procesador más bajo, ademas tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial, es aproximadamente 5.8 veces menor que el sistema que utiliza el mayor porcentaje de procesador que en este caso es Apache Spark. SciB es el segundo sistema con menor uso de procesador, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. HP vertica es el tercer sistema con menor uso de procesador, el cual tampoco tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras que Apache Spark tiene el uso de procesador más alto, este porcentaje no cambia cuando incrementa el porcentaje de error de integridad referencial.

### 4.5.11 Promedio por columna especificada

Los resultados obtenidos al momento de ejecutar la función para obtener el Promedio por columna especificada son los siguientes:

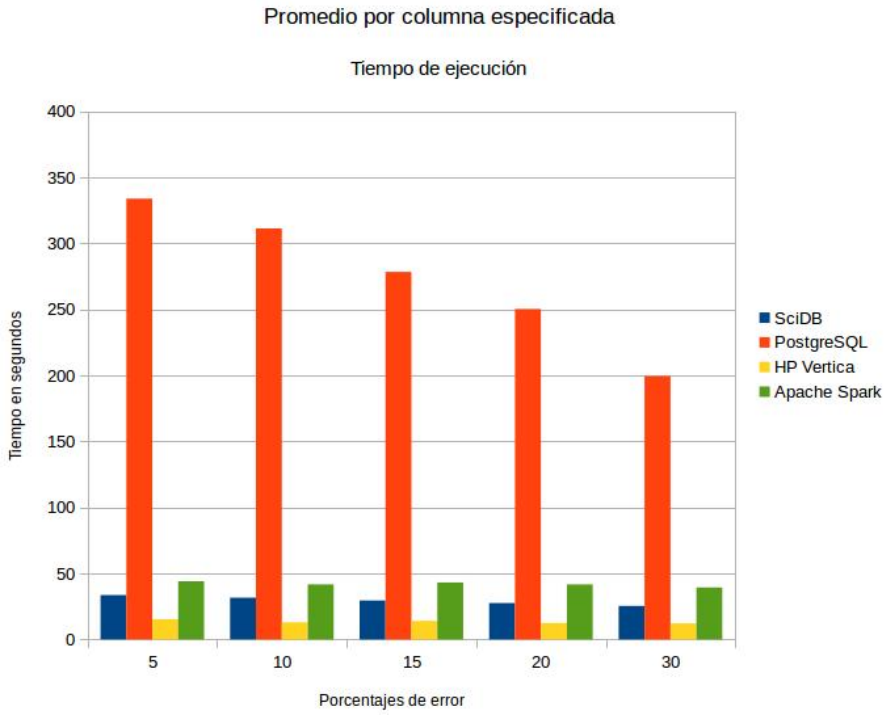


Figure 4.37: Promedio por columna especificada Tiempo de ejecución.

En el tiempo de ejecución se puede observar que PostgreSQL es el sistema con el mayor tiempo de ejecución y puede tardar aproximadamente 17 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 70% entre el tiempo más bajo y más alto que registró. Mientras que SciDB y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 20% entre el tiempo de ejecución más bajo y más alto que registró.

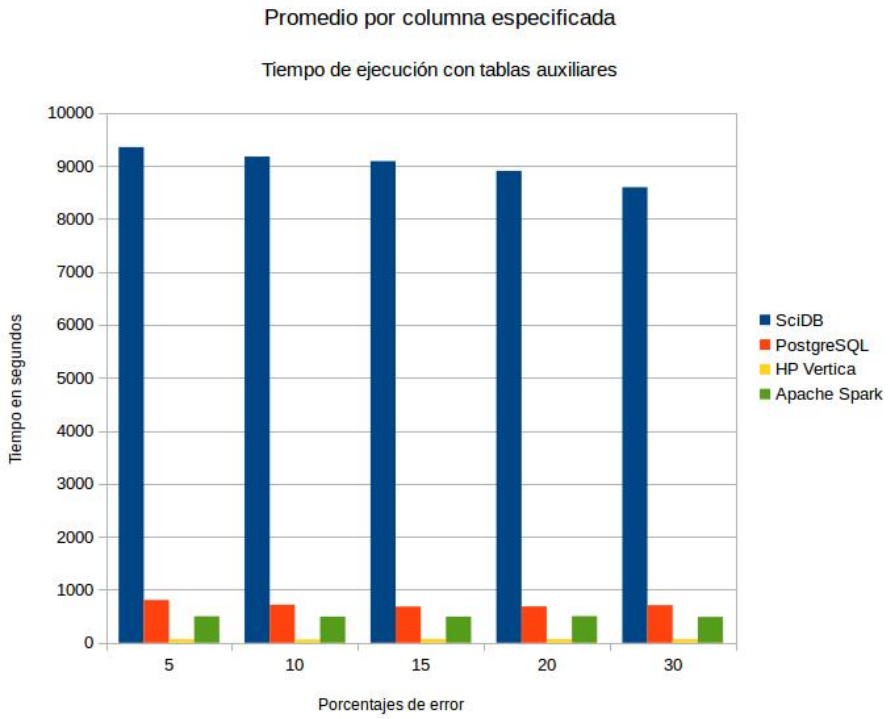


Figure 4.38: Promedio por columna especificada Tiempo de ejecución con tablas auxiliares.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es SciDB, seguido por PostgreSQL, después Apache Spark y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es HP Vertica.

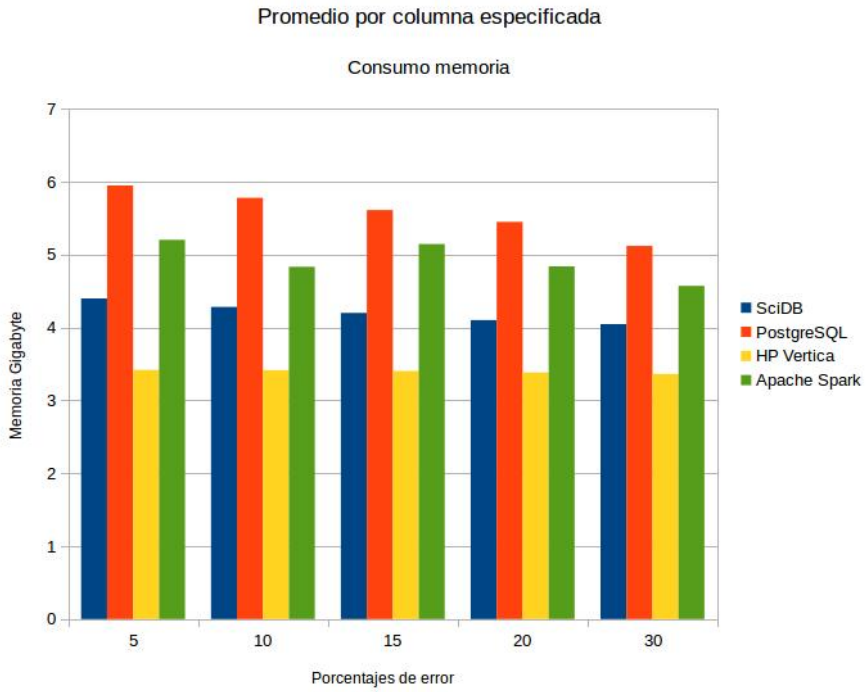


Figure 4.39: Promedio por columna especificada Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema que utiliza más memoria y consume aproximadamente 65% más memoria que el sistema que utiliza menos que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 15% comparado con el consumo más alto que registró. Apache Spark es el segundo sistema con el consumo de memoria más alto, no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras SciDB es el tercer sistema con el consumo de memoria más alto, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 10% comparado con el consumo más alto que registró. HP Vertica es el sistema que registra el menor consumo de memoria, manteniéndose igual sin importar el porcentaje de error de integridad referencial.

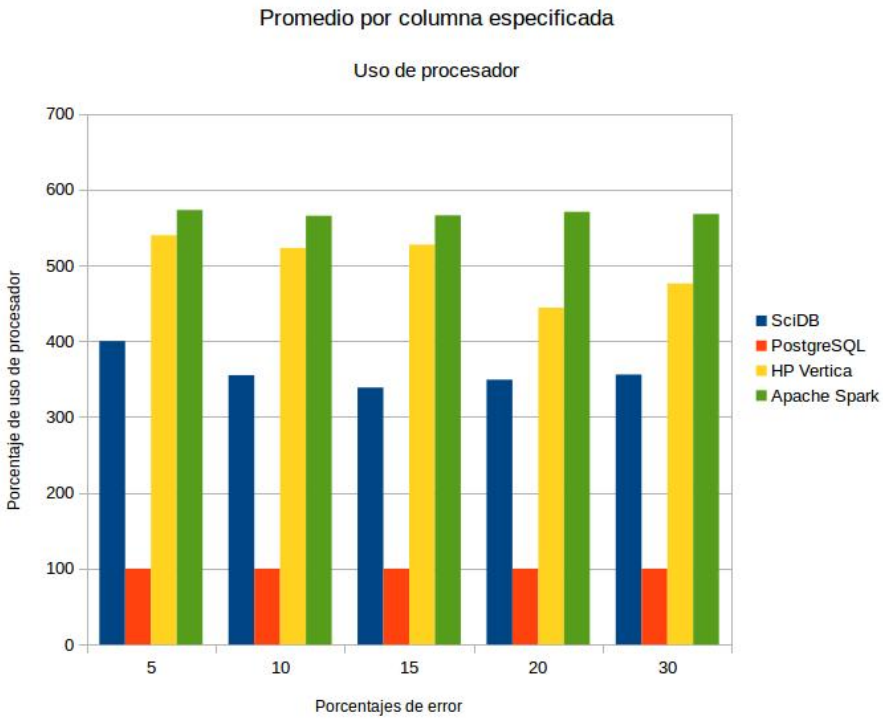


Figure 4.40: Promedio por columna especificada Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que PostgreSQL tiene el uso de procesador más bajo, ademas tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial, es aproximadamente 5.8 veces menor que el sistema que utiliza el mayor porcentaje de procesador que en este caso es es Apache Spark. SciB es el segundo sistema con menor uso de procesador, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. HP vertica es el tercer sistema con menor uso de procesador, el cual tampoco tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras que Apache Spark tiene el uso de procesador más alto, este porcentaje no cambia cuando incrementa el porcentaje de error de integridad referencial.

### 4.5.12 Promedio total por columna especificada

Los resultados obtenidos al momento de ejecutar la función para obtener el Promedio total por columna especificada son los siguientes:

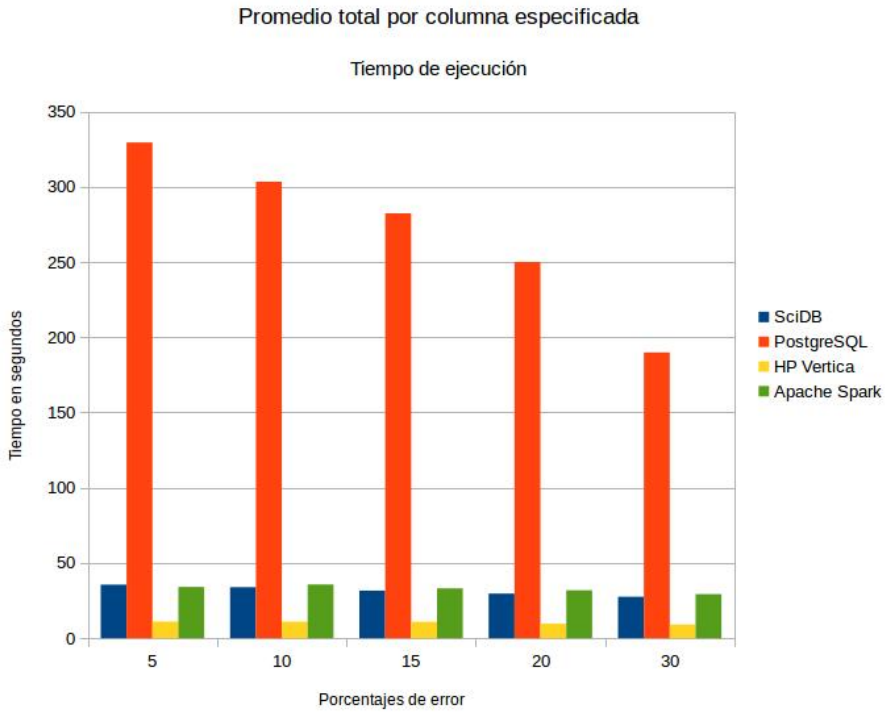


Figure 4.41: Promedio total por columna especificada Tiempo de ejecución.

En el tiempo de ejecución se puede observar que PostgreSQL es el sistema con el mayor tiempo de ejecución y puede tardar aproximadamente 16 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 70% entre el tiempo más bajo y más alto que registró. Mientras que SciDB y Apache Spark son muy parecidos, en cuanto a tiempo de ejecución y comportamiento. El sistema con el menor tiempo de ejecución es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede decrementar aproximadamente un 20% entre el tiempo de ejecución más bajo y más alto que registró.

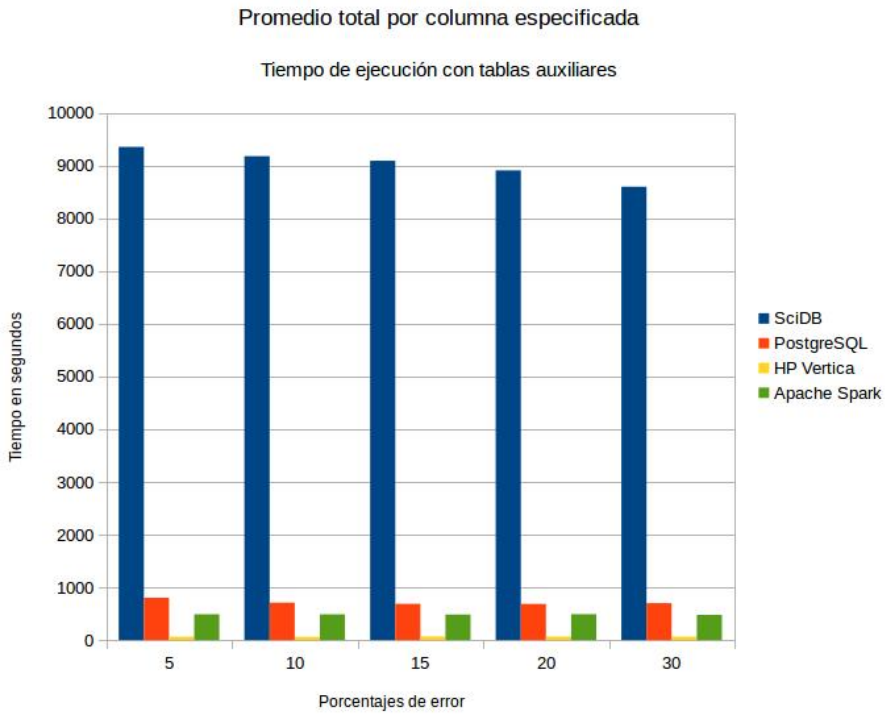


Figure 4.42: Promedio total por columna especificada Tiempo de ejecución con tablas auxiliares.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es SciDB, seguido por PostgreSQL, después Apache Spark y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es HP Vertica.

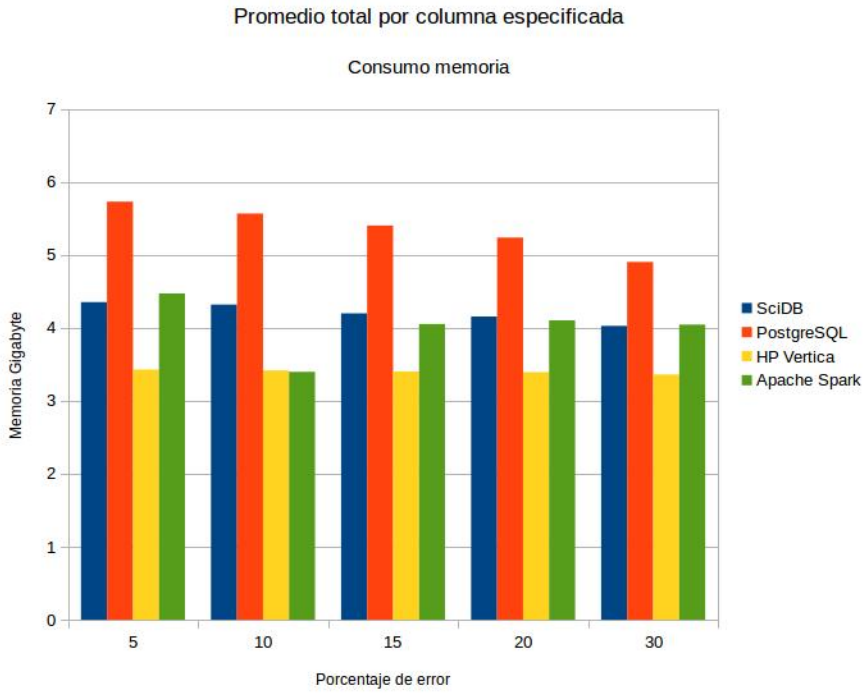


Figure 4.43: Promedio total por columna especificada Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema que utiliza más memoria y consume aproximadamente 60% más memoria que el sistema que utiliza menos que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 15% comparado con el consumo más alto que registró. SciDB es el segundo sistema con el consumo de memoria más alto, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede decrementar aproximadamente un 10% comparado con el consumo de memoria más alto que registró. Mientras Apache Spark es el tercer sistema con el consumo de memoria más alto, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. HP Vertica es el sistema que registra el menor consumo de memoria, manteniéndose igual sin importar el porcentaje de error de integridad referencial.

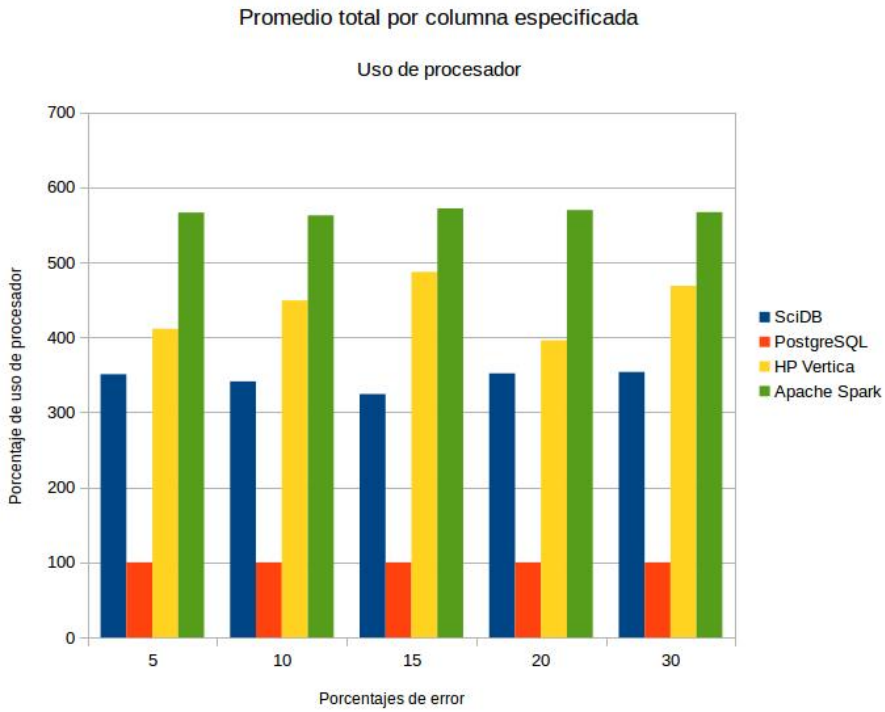


Figure 4.44: Promedio total por columna especificada Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que PostgreSQL tiene el uso de procesador más bajo, además tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial, es aproximadamente 5.8 veces menor que el sistema que utiliza el mayor porcentaje de procesador que en este caso es Apache Spark. SciB es el segundo sistema con menor uso de procesador, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. HP vertica es el tercer sistema con menor uso de procesador, el cual tampoco tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras que Apache Spark tiene el uso de procesador más alto, este porcentaje no cambia cuando incrementa el porcentaje de error de integridad referencial.

### 4.5.13 Valor máximo por columna especificada

Los resultados obtenidos al momento de ejecutar la función para obtener el Valor máximo por columna especificada son los siguientes:

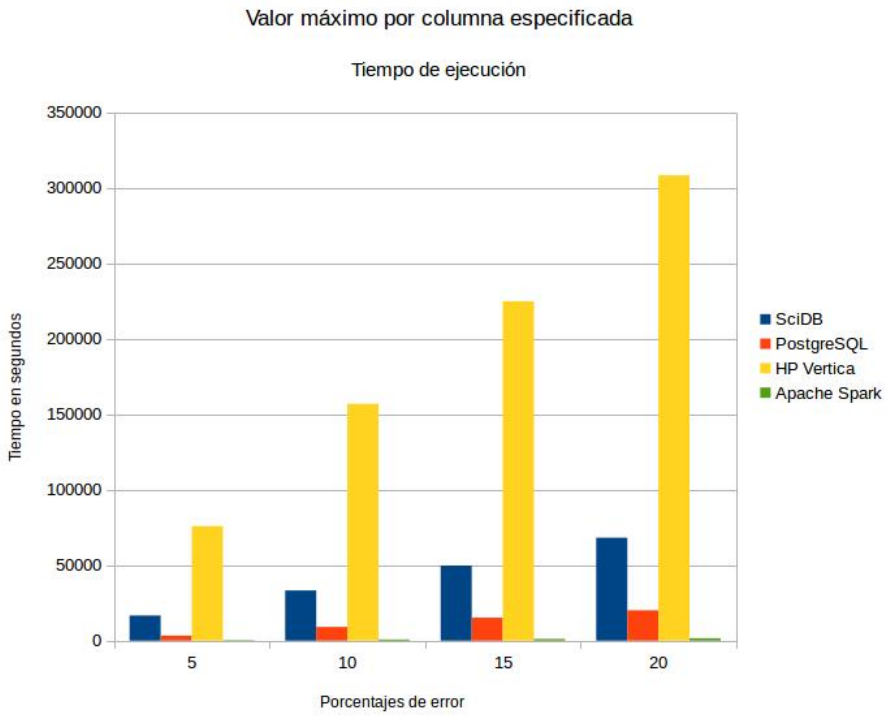


Figure 4.45: Valor máximo por columna especificada Tiempo de ejecución.

En el tiempo de ejecución se puede observar que HP Vertica es el sistema con el mayor tiempo de ejecución y puede tardar aproximadamente 193 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es Apache Spark, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 4.5 veces entre el tiempo más bajo y más alto que registro. Mientras que SciDB tiene el segundo lugar respecto al tiempo de ejecución, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 3 veces entre el tiempo más bajo y más alto que registro. El sistema que tiene el tercer lugar respecto al tiempo de ejecución es PostgreSQL, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 3 veces entre el tiempo más bajo y más alto que registro. El sistema con el menor tiempo de ejecución es Apache Spark, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 4 veces entre el tiempo de ejecución más bajo y más alto que registró.

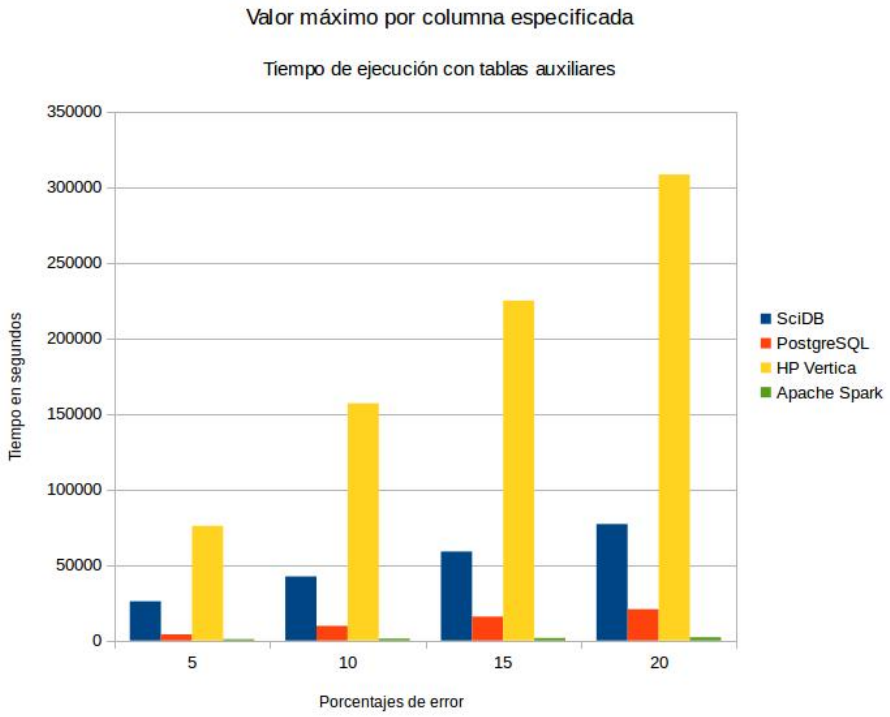


Figure 4.46: Valor máximo por columna especificada Tiempo de ejecución con tablas auxiliares.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es HP Vertica, seguido por SciDB, después PostgreSQL y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es Apache Spark.

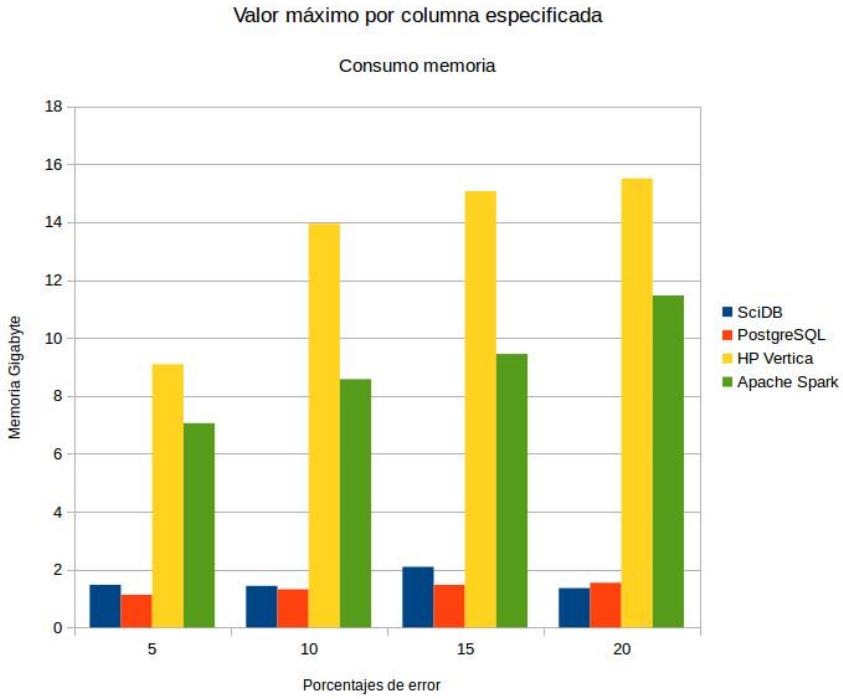


Figure 4.47: Valor máximo por columna especificada Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema que utiliza menos memoria y consume aproximadamente 8.5 veces menos memoria que el sistema que consume más memoria que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente un 25% comparado con el consumo más bajo que registro. SciDB es el segundo sistema con el consumo de memoria más bajo, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución. Mientras Apache Spark es el tercer sistema con el consumo de memoria más bajo, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente un 60% comparado con el consumo de memoria más bajo que registró. HP Vertica es el sistema que registra el mayor consumo de memoria, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente un 60% comparado con el consumo de memoria más bajo que registró.

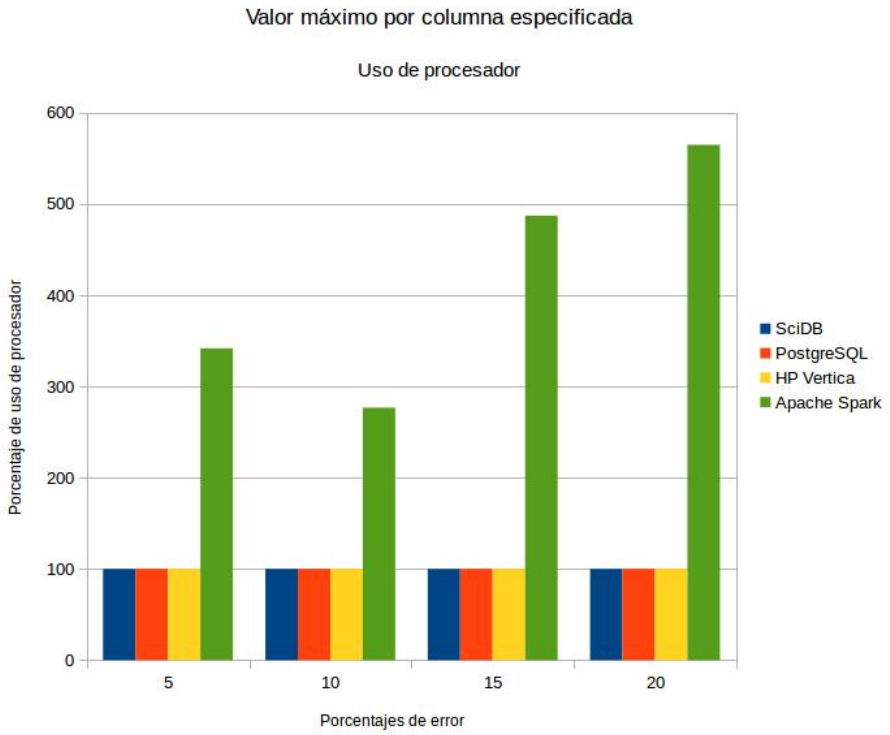


Figure 4.48: Valor máximo por columna especificada Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que SciDB, HP Vertica y PostgreSQL tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial. Mientras que Apache Spark es el sistema con el mayor uso de procesador, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución, puede llegar a ser hasta 5.8 veces mayor que los otros sistemas.

### 4.5.14 Valor mínimo por columna especificada

Los resultados obtenidos al momento de ejecutar la función para obtener el Valor mínimo por columna especificada son los siguientes:

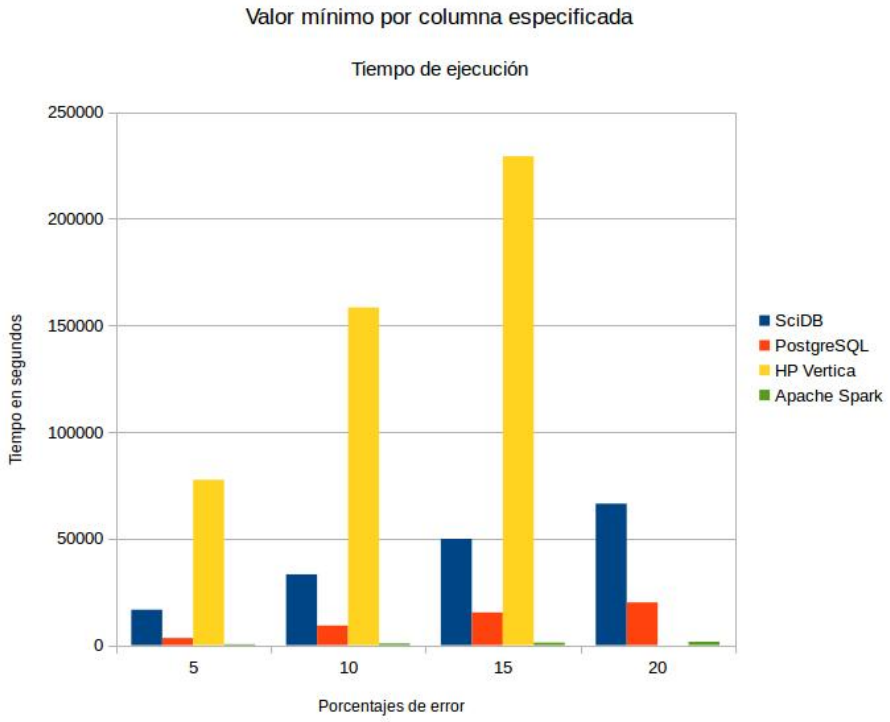


Figure 4.49: Valor mínimo por columna especificada Tiempo de ejecución.

En el tiempo de ejecución se puede observar que HP Vertica es el sistema con el mayor tiempo de ejecución y puede tardar en promedio aproximadamente 190 veces más tiempo que el sistema que tiene el menor tiempo de ejecución que en este caso es Apache Spark, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 4.5 veces, entre el tiempo más bajo y más alto que registró. Mientras que SciDB tiene el segundo lugar respecto al tiempo de ejecución,cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 3 veces entre el tiempo más bajo y ás alto que registró. El sistema que tiene el tercer lugar respecto al tiempo de ejecución es PostgreSQL, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 3 veces entre el tiempo más bajo y más alto que registró. El sistema con el menor tiempo de ejecución es Apache Spark, cuando incrementa el porcentaje de error de integridad referencial, su tiempo de ejecución puede incrementar aproximadamente 4 veces entre el tiempo de ejecución más bajo y más alto que registró.

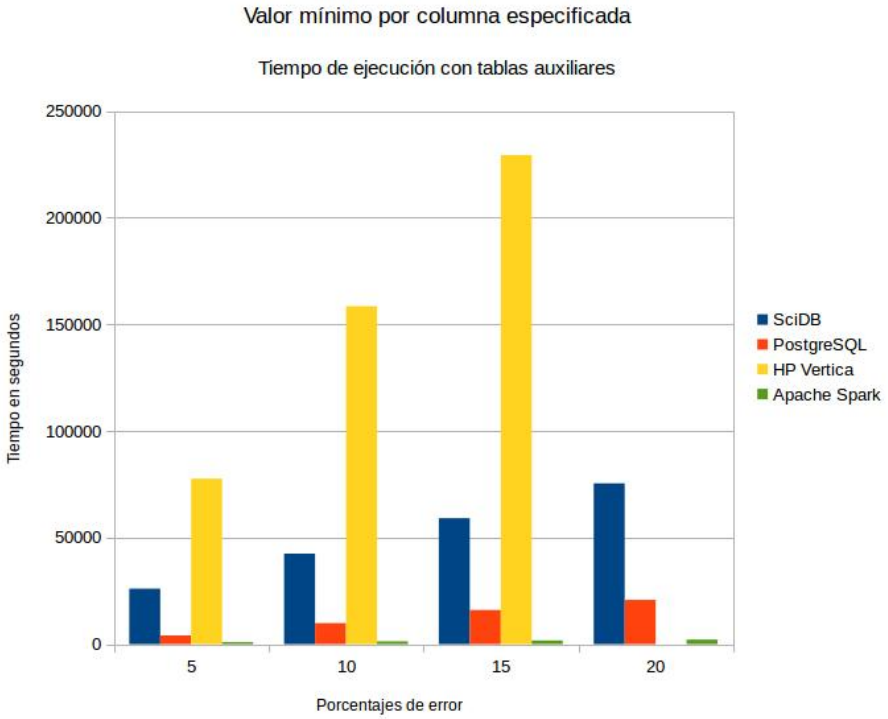


Figure 4.50: Valor mínimo por columna especificada Tiempo de ejecución con tablas auxiliares.

En cuanto al tiempo de ejecución tomando en cuenta la creación de tablas auxiliares se puede observar que el sistema que maneja los tiempos más altos es HP Vertica, seguido por SciDB, después PostgreSQL y por último el sistema con el menor tiempo de ejecución tomando en cuenta la creación de tablas auxiliares es Apache Spark.

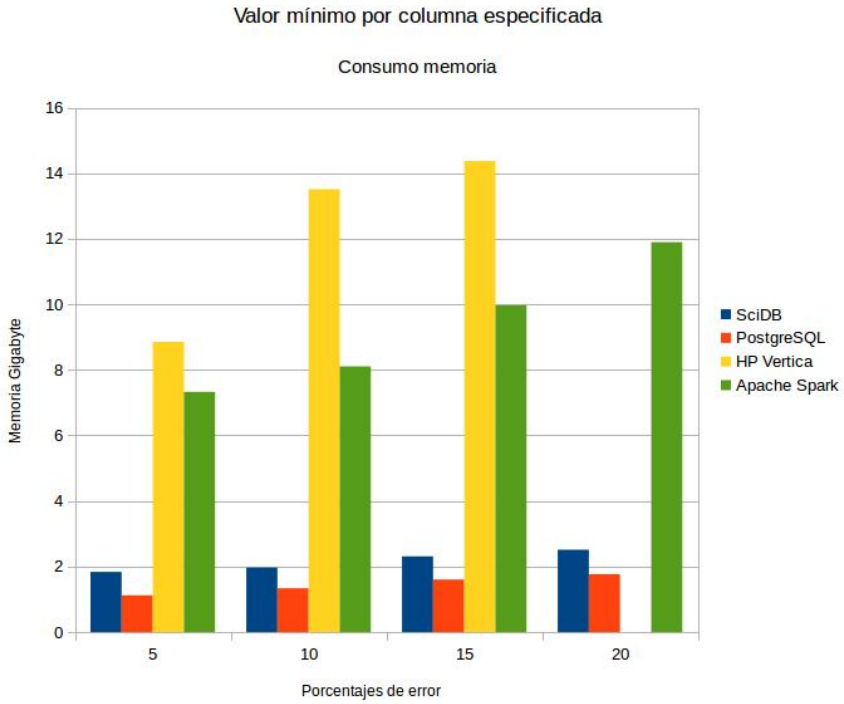


Figure 4.51: Valor mínimo por columna especificada Consumo de memoria.

En cuanto al consumo de memoria se puede observar que PostgreSQL es el sistema que utiliza menos memoria y consume aproximadamente 8 veces menos memoria que el sistema que consume más memoria que en este caso es HP Vertica, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente un 25% comparado con el consumo más bajo que registro. SciDB es el segundo sistema con el consumo de memoria más bajo, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente un 15% comparado con el consumo más bajo que registró. Mientras Apache Spark es el tercer sistema con el consumo de memoria más bajo, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente un 55% comparado con el consumo más bajo que registró. HP Vertica es el sistema que registra el mayor consumo de memoria, cuando incrementa el porcentaje de error de integridad referencial, su consumo de memoria puede incrementar aproximadamente un 60% comparado con el consumo de memoria más bajo que registró.

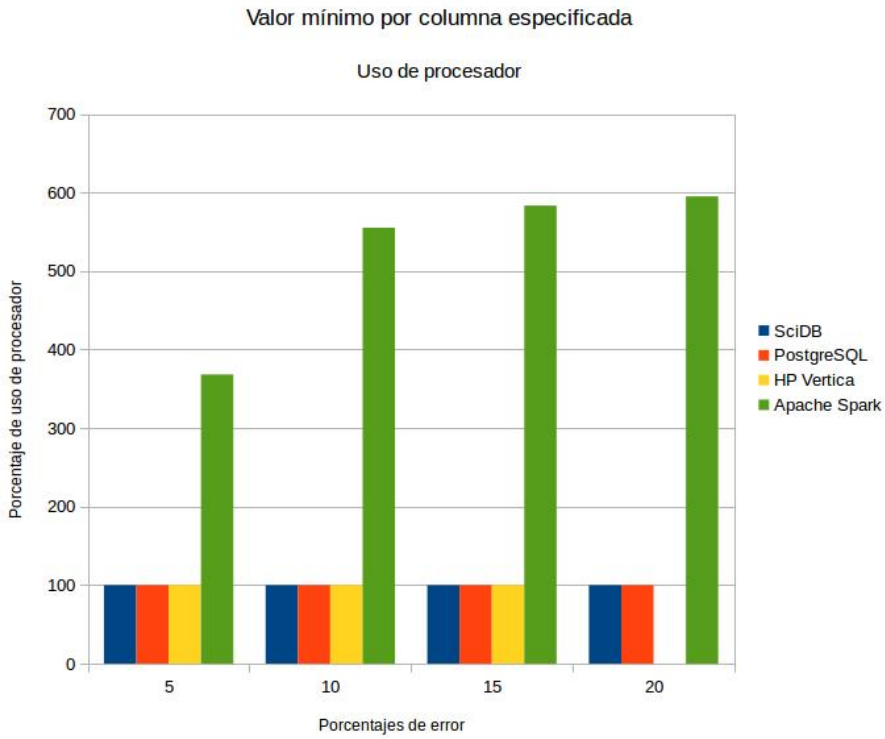


Figure 4.52: Valor mínimo por columna especificada Uso de procesador.

En cuanto al porcentaje de procesador se puede observar que SciDB, HP Vertica y PostgreSQL tiene un comportamiento constante sin importar el porcentaje de error de integridad referencial. Mientras que Apache Spark es el sistema con el mayor uso de procesador, el cual no tiene un comportamiento fijo; decrementa e incrementa en cada ejecución, puede llegar a ser hasta 5.9 veces mayor que los otros sistemas.

# CAPÍTULO 5

# Conclusiones y trabajo futuro

Se puede observar que en el caso de SciDB los tiempos de ejecución que manejo para la creación de las tablas auxiliares fueron los más altos, el segundo lugar en cuanto a tiempo de ejecución de las consultas para crear las tablas auxiliares lo tuvieron PostgreSQL y Apache Spark y por último el sistema en manejar los tiempos más pequeños en ejecución de consultas fue HP Vertica.

En cuanto al consumo de memoria los sistemas en manejar el consumo más alto de memoria al momento de ejecutar las consultas para la creación de las tablas auxiliares fueron ScidB y Apache Spark, el segundo sistema en manejar el consumo de memoria más alto fue PostgreSQL, y el sistema que consumió la menor cantidad de memoria fue HP vertica.

En cuanto al uso de procesador el sistema con el uso de procesador más bajo esta dividido, entre HP Vertica y PostgreSQL, el segundo sistema con el uso de procesador más bajo es Apache Spark y el sistema que registra el uso mas intenso de procesador es SciDB.

Para crear las tablas auxiliares en el cálculo de las Funciones de Agregación Extendida respecto al uso de procesador SciDB es el que tiene el uso más intensivo de procesador en la mayoria de las consultas seguido muy de cerca por Apache Spark los dos sistemas tienen los mejores porcentajes de uso de procesador, excepto en la última tabla que es la que tiene varias operaciones matemáticas que se guardan en las columnas que la conforman, ahí Apache Spark y HP Vertica manejan el uso más intensivo en cuanto al uso del procesador. El sistema que tiene los mejores tiempos de ejecución, el menor consumo de memoria RAM y el que almacena una menor cantidad de datos es HP Vertica.

SciDB maneja tiempos de ejecución altos debido a que no tiene integrado propiamente la reunión natural, así que se tuvo que buscar algún tipo de alternativa para poder simular esta reunión (ver sección 4.1).

Se puede observar que en el caso de PostgreSQL sus tiempos totales de ejecución de las consultas para las Funciones de Agregación Extendidas (excepto en las del cálculo del valor máximo y mínimo) es muy alta en comparación de los demás sistemas, pero con el incremento del porcentaje de error de integridad referencial, el tiempo de ejecución decrementa y para los demás sistemas el tiempo de ejecución se mantiene prácticamente igual.

El consumo de memoria RAM total para la ejecución de las consultas para las Funciones de Agregación Extendidas (excepto en las del cálculo del valor máximo y mínimo), con el incremento del porcentaje de error de integridad referencial, en el caso de PostgreSQL y SciDB decrementa el consumo de memoria, en el caso de HP Vertica se mantiene igual con todos los porcentajes, y Apache Spark no tiene un comportamiento definido, decrementa e incrementa en diferentes ocasiones.

El porcentaje de uso de procesador para la ejecución de las consultas para las Funciones de Agregación Extendidas (excepto en las del cálculo del valor máximo y mínimo), con el incremento del porcentaje de error de integridad referencial, en el caso de PostgreSQL se mantiene igual, en el caso de Apache Spark hay muy poca variación en el uso de procesador siempre se mantiene alto, y para el caso de SciDB y HP Vertica sus resultados no se mantienen definidos, decrementan e incrementan en diferentes ocasiones.

Después de haber obtenido los resultados de la ejecución de las consultas en los diferentes sistemas, y analizados los mismos se puede llegar a las siguientes conclusiones. En el cálculo de las Funciones de Agregación Extendida excepto en las del cálculo del valor máximo y mínimo, el menor tiempo de ejecución de las consultas y el menor uso de memoria RAM lo tiene HP Vertica, en cuanto a los sistemas que utilizan mayor porcentaje de procesador durante la ejecución de las consultas esta dividido en dos sistemas: Apache Spark, seguido muy de cerca de HP Vertica. Analizando los datos el sistema con un mejor rendimiento en el cálculo de las Funciones de Agregación Extendida, exceptuando el cálculo del valor mínimo y el máximo, es HP Vertica, esto dado a que sus tiempos de ejecución son muy bajos, su poco consumo de memoria RAM y su intenso uso de procesador.

El desempeño superior de HP Vertica, se debe principalmente a su almacenamiento columnar, esto optimiza el acceso a los datos ya que la información se almacena en la misma forma en la que se consulta, las operaciones de lectura se ven beneficiadas ya que se reduce el numero de accesos al disco, esto en comparación al almacenamiento basado en filas de las bases de datos relacionales. Además de su codificación agresiva y su compresión, permite que tenga un rendimiento analítico que reduzca el uso de CPU, memoria RAM y las lecturas de disco en el tiempo de procesamiento, la reducción de datos puede ser hasta 1/10 de su tamaño original. Mientras que el tiempo de ejecución de PostgreSQL es muy superior debido a que tienen que leer todas las columnas para poder hacer las consultas, el uso de memoria alto es porque no tiene una buena compresión. El acceso a los datos en los arreglos de SciDB es rápido y la compresión de los mismos es buena, esto se ve reflejado en el uso de memoria, debido a sus 4 instancias que trabajan en conjunto el uso de procesador es muy bueno. En cuanto a Apache Spark comparte muchas similitudes con HP Vertica, el acceso a datos es de los más rápidos, la compresión de los datos se ve reflejado en un uso de memoria bajo, y el uso de procesador es el más intensivo de todos, tomando en cuenta que este sistema es para análisis de big data.

El tiempo para la ejecución de las consultas para las Funciones de Agregación Extendidas para el cálculo del valor máximo y mínimo, con el aumento del porcentaje de error de integridad referencial, en el caso de PostgreSQL es bajo e incrementa poco. En el caso de SciDB es bajo y también incrementa poco. En el caso de HP Vertica es alto e incrementa mucho. Y para Apache Spark es el que menor tiempo de ejecución tiene e incrementa muy poco.

El consumo de memoria RAM para la ejecución de las consultas para las Funciones de Agregación Extendidas para el cálculo del valor máximo y mínimo, con el incremento del porcentaje de error de integridad referencial, en el caso de PostgreSQL es bajo e incrementa poco. En el caso de SciDB es bajo e incrementa poco. En el caso de HP Vertica es alto e incrementa considerablemente. Y para Apache Spark tiene un consumo alto e incrementa.

El porcentaje de uso de procesador para la ejecución de las consultas para las Funciones de Agregación Extendidas para el cálculo del valor máximo y mínimo, con el incremento del porcentaje de error de integridad referencial, PostgreSQL, SciDB y HP Vertica se mantienen igual. Y para Apache Spark sus resultados no se mantienen definidos, decrementan e incrementan en diferentes ocasiones.

Para el cálculo de los valores máximo y mínimo utilizando las Funciones de Agre-

gación Extendida correspondiente, como se explicó en secciones anteriores SciDB y HP Vertica no soportaron los cálculos matemáticos necesarios y se utilizaron programas auxiliares (R y Python respectivamente) para el calculo de estas funciones en específico. Con esto, el sistema en manejar los tiempos más bajos en cuanto a la ejecución de las consultas, por una gran ventaja fue Apache Spark. En cuanto al uso de memoria RAM el sistema que utilizó la menor cantidad de memoria fue PostgreSQL. Y por último en cuanto al uso de procesador, el sistema que tuvo un mejor uso de éste fue Apache Spark. Analizando los datos el sistema con un mejor rendimiento en el cálculo de las Funciones de Agregación Extendida para el cálculo del valor máximo y mínimo, es Apache Spark por su tiempo de ejecución extremadamente bajo, un uso de memoria RAM moderado, su intenso uso de procesador.

Apache Spark fue superior por el hecho de no necesitar ninguna herramienta extra para obtener los resultados requeridos, al realizar los cálculos dentro del mismo sistema el acceso a los datos necesarios es superior, eso hace que los tiempos bajen considerablemente . Mientras que PostgreSQL también tuvo un desempeño superior de igual manera que Apache Spark debido a que todos los proceso se hicieron dentro de la misma función esto se vio reflejado en un bajo tiempo de ejecución, en cuanto consumo de memoria tuvo un buen desempeño, a pesar de no ser su fuerte los cálculos matemáticos se pudo mantener como el segundo mejor. En cuanto a SciDB se vio afectado sus tiempos ya que los cálculos sobre los datos los realizó R, esto repercutió en gran medida en el tiempo de ejecución, aunque la compresión de datos es buena y se vio reflejado en el uso de memoria, aunque el uso de procesador de R es inferior al que maneja SciDB y esto se vio afectado al analizar esta características. Mientras que HP Vertica tuvo el peor desempeño de todos al igual que SciDB los cálculos de las funciones se hicieron por fuera en Python, solo se extrajeron los datos de HP Vertica, esto se vio reflejado en su tiempo de ejecución, un consumo de memoria alto por no contar Python con la misma codificación y compresión que HP Vertica, al igual que un uso de procesador inferior, todas estas características lo hicieron pasar del primer lugar al último.

La linea de investigación para realizar un trabajo futuro, es realizar estas implementaciones en servidores de múltiples nodos en forma de una implementación en paralelo ya sea en forma centralizada o distribuida (aquí interferiría como factor importante la red), para poder observar los cambios que hay tanto en la implementación, en las variables y los resultados, debido a que estos experimentos se realizaron en un servidor de un único nodo.

Otras opciones para trabajo a futuro seria realizar estas implementaciones en una infraestructura diferente, como podría ser bases de datos en memoria con un solo nodo, y con múltiples nodos, para esto se necesitaría un servidor con características diferentes al que utilizo en esta implementación.

# BIBLIOGRAFÍA


[1] J. García-García and C. Ordonez. Extended aggregations for databases with referential integrity issues. *Data & Knowledge Engineering*, 69:73–95, 2010.

[2] ISO-ANSI. *Database Language SQL-Part2: SQL/Foundation*, ansi, iso 9075-2 edition, 1999.

[3] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE TKDE*, 15(6):1389–1408, 2003.

[4] O. Arieli, M. Denecker, B. Van Nuffelen, and M. Bruynooghe. Database repair by signed formulae. *FoIKS*, 2942:14–30, 2004.

[5] J. Wijsen. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3):722–768, 2000.

[6] Tpc.org. `http://www.tpc.org/tpch/`, 2017.

[7] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. *ACM SIGMOD Conference*, page 240–251, 2002.

[8] E. Rahm and H. Do Hong. Data cleaning: problems and current approaches. *IEEE Bull. Tech. Committee Data Eng.*, 23(4), 2000.

[9] S. McClean, B. Scotney, and M. Shapcott. Aggregation of imprecise and uncertain information in databases. *IEEE TKDE*, 13(6):902–912, 2001.

[10] J. García-García and C. Ordonez. Consistent aggregations in databases with referential integrity errors. *ACM International Workshop on Information Quality in Information Systems*, IQUIS:80–89, 2006.

[11] J. García-García and C. Ordonez. Estimating and bounding aggregations in databases with referential integrity errors. *ACM DOLAP Workshop*, pages 49–56, 2008.