
# Catálogo de arquitecturas de software y tácticas arquitectónicas para contextos de big data

Russo, Juan Pablo
*Universidad ORT Uruguay*

Solari, Martín
*Universidad ORT Uruguay*

Mousqués, Gastón
*Universidad ORT Uruguay*

Setiembre, 2019

**Abstract**

El presente documento presenta un catálogo de arquitecturas de software y tácticas arquitectónicas aplicables en contextos de big data. En la primera parte se describen las arquitecturas de software utilizando un esquema que presenta sus escenarios de uso y principales componentes. A partir de estas descripciones se identificaron tácticas arquitectónicas comunes aplicadas en contextos de big data. Se presenta una descripción de cada táctica arquitectónica y su estrategia de resolución de los atributos de calidad afectados.

**Palabras clave:** big data, arquitectura de software, tácticas arquitectónicas.

Documento de Investigación, Nro.15, setiembre 2019. Universidad ORT Uruguay.
Facultad de Ingeniería. ISSN 1688-8707.

# Catálogo de arquitecturas de software y tácticas arquitectónicas para contextos de big data

Juan Pablo Russo, Martín Solari, Gastón Mousqués

Universidad ORT Uruguay
jprusso@uni.ort.edu.uy, martin.solari@ort.edu.uy, mousques@uni.ort.edu.uy

**Resumen.** El presente documento presenta un catálogo de arquitecturas de software y tácticas arquitectónicas aplicables en contextos de big data. En la primera parte se describen las arquitecturas de software utilizando un esquema que presenta sus escenarios de uso y principales componentes. A partir de estas descripciones se identificaron tácticas arquitectónicas comunes aplicadas en contextos de big data. Se presenta una descripción de cada táctica arquitectónica y su estrategia de resolución de los atributos de calidad afectados.

**Palabras clave:** big data, arquitectura de software, tácticas arquitectónicas.

# Contenido


1 Introducción ..... 4
2 Arquitecturas de software para big data ..... 5
2.1 Arquitecturas industriales ..... 5
2.1.1 Arquitecturas de big data en LinkedIn ..... 5
2.1.2 Arquitecturas de big data en Twitter ..... 7
2.2 Arquitecturas teóricas ..... 8
2.2.1 Arquitectura lambda ..... 9
2.2.2 Arquitectura kappa ..... 10
2.2.3 Arquitectura SOLID ..... 10
2.3 Arquitecturas de referencia ..... 12
2.3.1 Arquitectura de Maier ..... 13
2.3.2 Arquitectura de Pääkkönen ..... 15
2.3.3 Arquitectura de Geerdink ..... 15
2.3.4 Arquitectura de Viana ..... 16
3 Tácticas arquitectónicas ..... 17
3.1 Escalabilidad ..... 17
3.1.1 Escenarios de uso general de escalabilidad en big data ..... 18
3.1.2 Particionamiento de datos ..... 18
3.1.3 Entorno share-nothing ..... 19
3.1.4 Procesamiento por lotes ..... 20
3.2 Eficiencia ..... 22
3.2.1 Escenarios de uso general de eficiencia en big data ..... 23
3.2.2 Consistencia eventual ..... 23
3.2.3 Procesamiento en tiempo real ..... 25
3.3 Modificabilidad ..... 25
3.3.1 Escenarios de uso general de modificabilidad en big data ..... 26
3.3.2 Persistencia políglota ..... 26
3.4 Disponibilidad ..... 28
3.4.1 Escenarios de uso general de disponibilidad en big data ..... 29
3.4.2 Write-Ahead Log ..... 29
3.4.3 Semánticas de llamadas remotas ..... 30

3.4.4 Replicación de datos y funcional .....31
4 Referencias bibliográficas.....34

# 1 Introducción

Este documento presenta un extracto en forma de catálogo de arquitecturas de software y tácticas arquitectónicas, analizadas en una tesis de Maestría en Ingeniería [1]. El propósito del catálogo es facilitar al practicante las decisiones arquitectónicas en contextos de big data.

Se presenta la descripción de un subconjunto de arquitecturas de software de big data identificadas previamente en un estudio de mapeo sistemático [2]. Se seleccionaron arquitecturas de software para las que se cuenta con un nivel de documentación suficiente y dicha documentación se encuentra disponibles públicamente. Para cada arquitectura se plantean los escenarios de uso, un diagrama de alto nivel de la solución arquitectónica y la descripción de sus principales componentes y capas.

A partir de la descripción de las arquitecturas se identificaron las tácticas arquitectónicas comunes. Se presenta la descripción de cada táctica junto con las estrategias de resolución para los atributos de calidad afectados.

# 2 Arquitecturas de software para big data

En el estudio de mapeo sistemático [2] se detectaron 90 propuestas de arquitecturas de software de big data clasificadas en tres tipos desde el punto de vista de su contexto de aplicación.

- **Industriales**: Describen soluciones de aplicación corporativas de big data dentro del contexto de una organización en particular.
- **Teóricas**: Describen soluciones más generales en diferentes contextos pero restringidas a algún problema particular dentro del área de big data.
- **Referencia**: Describen arquitecturas de referencia que capturan el conocimiento de otras arquitecturas existentes y sirven de guía para los practicantes en sus futuros diseños arquitectónicos.

A continuación, se describen las arquitecturas de software más representativas de cada tipo encontradas en la literatura.

La descripción de las arquitecturas se presenta con la siguiente estructura general.

- **Contexto de uso**: En una primera instancia se resume el contexto de uso de la arquitectura para comprender mejor su contexto de aplicación. Esto puede incluir los requerimientos y escenarios de uso propios del negocio (en los casos de arquitecturas industriales) o el problema general que tratan de resolver (en el caso de arquitecturas teóricas).
- **Descripción arquitectura**: La descripción contiene un diagrama que resume visualmente las principales capas y componentes de la arquitectura. El diagrama se acompaña de una descripción más detallada de cada uno de sus componentes y principales interacciones. En los casos que apliquen pueden mencionarse las tecnologías utilizadas o sugeridas para la implementación de la solución.

## 2.1 Arquitecturas industriales

Existen varias empresas que necesitan resolver problemáticas de big data bajo sus contextos organizacionales particulares y publican aspectos de su solución arquitectónica. A continuación, se describen las arquitecturas de software de las organizaciones encontradas.

### 2.1.1 Arquitecturas de big data en LinkedIn

**LinkedIn** analiza los datos generados por sus sitios y aplicaciones para derivar la visión e ideas de sus nuevas funcionalidades. El sistema debe manejar alrededor de 100 TB de datos comprimidos en aproximadamente 300 tópicos diferentes. Dicha infraestructura debe procesar unos 15 billones de mensajes por día con picos

sostenidos de hasta 200.000 mensajes por segundo. Su arquitectura [3] utiliza un conjunto de sistemas de mensajería distribuidos [4] con procesamiento en lotes [5] coordinados por un planificador de tareas. La Figura 1 muestra la arquitectura de LinkedIn utilizada por sus analistas de datos para procesar la información de sus sistemas.

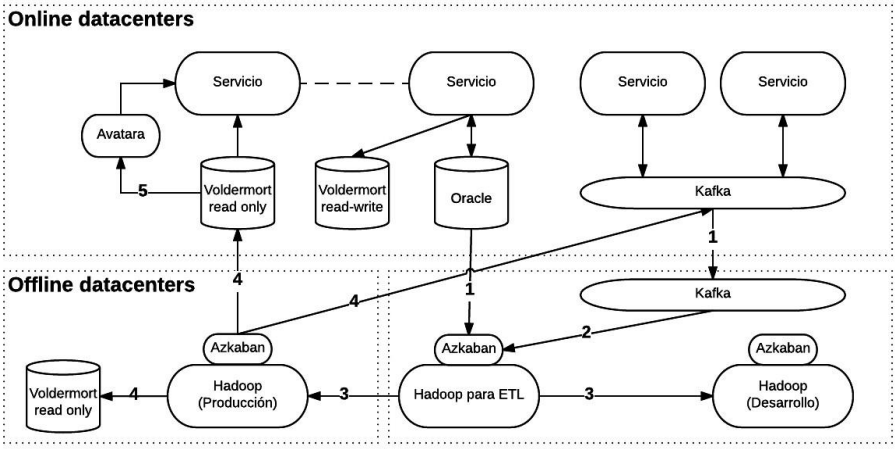


Figura 1 Arquitectura de LinkedIn [3]

1. Los datos son recolectados de dos orígenes principales:
    a. Snapshots de la base de datos sobre información de los usuarios, compañías, conexiones y otros datos primarios del sitio.
    b. Las actividades de los usuarios que se generan como un flujo de eventos basados en el uso de los servicios de LinkedIn.
2. Kafka [4] es un sistema de mensajería (productor-consumidor) distribuido que es usado para recolectar el flujo de eventos de los usuarios. Los productores reportan los eventos agrupados por tópicos a un broker. De esta forma los consumidores pueden leer los eventos a su propio ritmo. La información de los eventos son transferidos a un cluster de Hadoop [5] ETL para su posterior procesamiento.
3. Los datos en el cluster de Hadoop ETL son replicados a clusters de producción y desarrollo. Azkaban es usado como planificador de trabajo soportando una diversidad de tipos de trabajos (MapReduce, Pig, Shell scripts y Hive). Típicamente los trabajos son probados en el cluster de desarrollo antes de ser puestos en producción.
4. Los resultados de los análisis en el ambiente de producción son transferidos a una base de datos para facilitar su depuración frente a posibles incidentes. Los resultados pueden ser devueltos al cluster de Kafka para ser consumidos por los servicios de la aplicación.

5. Avatara [6] es utilizado para la preparación de datos de OLAP. Los datos analizados son leídos desde una base de datos Voldemort para su preprocesamiento, agregación y cubificación de OLAP para finalmente ser persistidos en una nueva base de datos Voldermort.

Para manejar las conexiones de su red social se utiliza un servicio de miembros de la plataforma basado en grafos distribuidos [7]. Para el contenido del sitio se utiliza una base relacional de Oracle, que fue migrada posteriormente a una fuente única de datos documental bajo Espresso [8].

### 2.1.2 Arquitecturas de big data en Twitter

**Twitter** es una plataforma de comunicación donde sus usuarios pueden escribir mensajes de 280 caracteres de largo llamados "tweets" a sus seguidores (otros usuarios que se subscriben a dichos mensajes). Twitter tiene más de 100 millones de usuarios activos mundialmente, que colectivamente envían más de 500 millones de tweets diarios para un histórico de más de 300 billones de tweets desde el 2006. Para la búsqueda de tweets la aplicación cuenta con un buscador que debe servir más de 2 billones de consultas diarias con una latencia promedio de 50 ms. Su arquitectura [9] fue diseñada para que los tweets puedan ser encontrados luego de 10 segundos de ser creados con procesamientos en tiempo real. Existe el desafío adicional de guardar los patrones de uso [10] de los usuarios para su posterior análisis [11] bajo procesamiento por lotes. La Figura 2 muestra la arquitectura del buscador de tiempo real de Twitter [9]. La arquitectura original estaba basada en el stack de Hadoop para su procesamiento en lotes. Sin embargo, los requerimientos de tiempo real de la búsqueda exigían tiempos de latencia que no eran alcanzados por la configuración original.

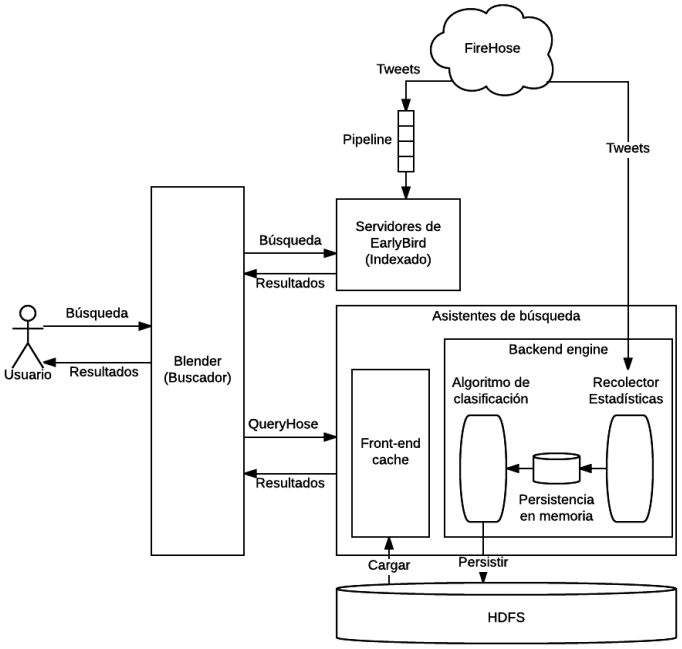


Figura 2 Arquitectura del buscador de tiempo real de Twitter [9]

En la nueva arquitectura de Twitter existe un front-end denominado Blender que recibe todas las búsquedas de la plataforma. Las consultas incluyen la búsqueda de tweets y cuentas de usuario a través de un servicio denominado QueryHose. Los tweets son creados por otro servicio ("FireHose") dirigidos a un pipeline para su anotación y conversión. Luego dichos tweets son enviados a los servidores de EarlyBird para su filtrado, personalización e indexado. EarlyBird [12] es un motor de búsqueda de tiempo real diseñado para proveer consultas con baja latencia y alto rendimiento. La arquitectura cuenta con asistentes de búsqueda adicionales como recolectores de estadísticas y algoritmos para la clasificación de los tweets. Estos análisis son persistidos en el sistema de HDFS de Hadoop para su posterior uso por los servicios de la plataforma.

Existen otras arquitecturas organizacionales en la literatura como **Facebook** [13] y **Netflix** [14] que demuestran los desafíos de escalabilidad, disponibilidad y performance de los sistemas de big data [15].

## 2.2 Arquitecturas teóricas

En este apartado se describen las arquitecturas teóricas de big data encontradas en la literatura que proponen soluciones más generales aplicables a un mayor número de contextos.

### 2.2.1 Arquitectura lambda

La arquitectura **Lambda** [16] es una arquitectura de procesamiento de datos, diseñada para gestionar un volumen masivo de datos, combinando el procesamiento por lotes (batch-processing) con el procesamiento de flujo de datos en tiempo real (stream-processing) [17]. Esta aproximación permite balancear la latencia, escalabilidad y disponibilidad necesarias para una solución de big data. La escalabilidad y precisión de las consultas se desarrolla con el procesamiento por lotes, mientras que la baja latencia y disponibilidad se construye con el procesamiento de datos en tiempo real. Esto permite alcanzar la consistencia eventual [18] mientras el desfasaje de la capa por lotes (batch layer) se sincroniza con la capa de tiempo real (speed layer). La Figura 3 muestra las capas que componen la arquitectura lambda. Las principales propiedades y relaciones entre capas se resumen a continuación [19].

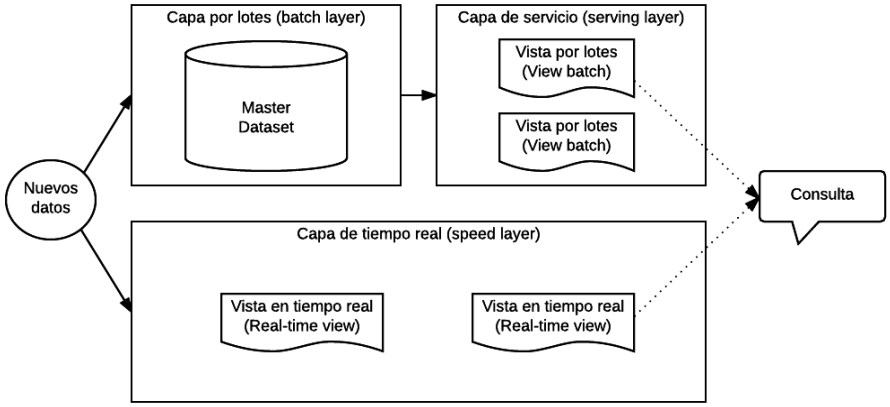


Figura 3 Arquitectura Lambda

1. Todos los datos que ingresan al sistema son despachados a la capa por lotes (batch layer) y a la capa de flujo en tiempo real (speed layer) simultáneamente para su procesamiento.
2. La capa de lotes (batch layer) tiene dos funciones principales:
    a. Gestionar el conjunto de datos maestros (master dataset). Es un conjunto de datos inmutables representando toda la información del sistema. Una vez almacenados no pueden ser alterados.
    b. Gestionar el preprocesamiento y cálculo de las vistas por lotes (batch views).
3. La capa de servicio (serving layer) indexa las vistas por lotes (batch views) para que puedan ser consultadas con baja latencia y alta velocidad de acceso.
4. La capa de flujo en tiempo real (speed layer) compensa la alta latencia de actualizaciones y sincronización de la capa de servicio (serving layer) procesando únicamente los datos recientes del sistema.

5. Toda consulta del sistema puede ser resuelta combinando los resultados de las vistas de lotes (batch views) y flujo en tiempo real (real-time views).

### 2.2.2 Arquitectura kappa

La arquitectura **Kappa** [20] es una simplificación de la arquitectura Lambda. Utiliza un log inmutable pero remueve la complejidad de la capa de procesamiento por lotes. Para lograr esta simplificación utiliza una única capa de procesamiento de tiempo real. Sin embargo, está capa puede ir evolucionando con nuevas lógicas de procesamiento necesarias para servir nuevos resultados. Para esto se utiliza el log como almacenamiento canónico de los datos originales y se promueven nuevas versiones de procesamiento de los datos. Se utiliza un sistema de versionado para ir liberando las nuevas actualizaciones del procesamiento en producción como se muestra en la Figura 4.

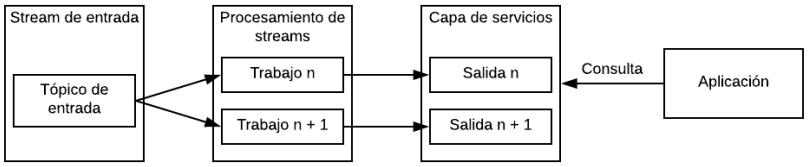


Figura 4 Arquitectura Kappa

### 2.2.3 Arquitectura SOLID

La arquitectura SOLID (Service-OnLine-Index-Data) [21] surge para resolver problemas de big data en sistemas de tiempo real. La arquitectura se compone de capas que separan la complejidad de gestionar grandes volúmenes de datos con respecto a su generación y consumo en tiempo real.

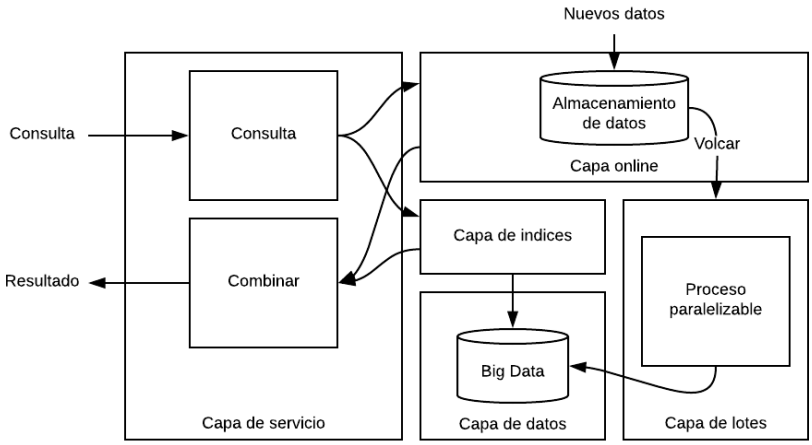


Figura 5 Arquitectura SOLID [21]

La Figura 5 muestra las diferentes capas de la arquitectura que se resumen a continuación.

- **Capa de tiempo real (Online Layer):** La capa superior resuelve las necesidades de acceso en tiempo real de los datos de entrada. Los nuevos datos son capturados en almacenamientos temporarios de alta velocidad de acceso. Estos repositorios son utilizados como un gran buffer para generar y consumir los datos recientes de la aplicación (todos aquellos datos que no existen todavía en el almacenamiento principal).
- **Capa de datos (Data Layer):** La capa de datos contiene el almacenamiento principal diseñado para almacenar grandes volúmenes de datos, su organización y semántica. Este almacenamiento puede tener alta latencia y tiempos de respuesta lentos.
- **Capa de índices (Index Layer):** Esta capa provee los índices para la capa de datos (Data Layer) con una alta disponibilidad y velocidad de acceso. Los índices son generados con la semántica de los datos de la capa inferior con el objetivo de proveer un rápido acceso para las consultas necesarias de la aplicación.
- **Capa de servicios (Service Layer):** La capa de servicios se presenta como una fachada para el usuario externo de la aplicación. Todas las consultas son realizadas a través de esta capa que redirige las consultas a las capas de tiempo real e índices para resolverlas y combinarlas para conseguir la respuesta final.
- **Capa de combinación (Merge Layer):** La capa de combinación recibe y transforma los datos de entrada para combinarlos con los datos existentes en el almacenamiento principal. Esto implica generalmente un procesamiento

intenso de datos por lo que debe ser procesado en lotes usando alguna técnica de paralelización como puede ser MapReduce [22].

## 2.3 Arquitecturas de referencia

Durante los años recientes surgen arquitecturas de referencia [23] [24] [25] [26] que tratan de abstraer y capturar el conocimiento de las arquitecturas teóricas y concretas de la industria. Angelov [27] clasifica las arquitecturas de referencia en 5 tipos resumidos en la Tabla 1.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Porque** | Estandarización | Estandarización | Facilitación | Facilitación | Facilitación |
| **Cuando** | Múltiples organizaciones | Única organización | Múltiples organizaciones | Única organización | Múltiples organizaciones |
| **Quién** | Organizaciones de estandarización, usuarios o arquitectura | Grupo de usuarios o arquitectos | Organización independiente de software o usuarios | Grupo de usuarios o arquitectos | Centros de investigación, organizaciones de arquitectura o usuarios |
| **Cuando** | Clásica | Clásica | Clásica | Clásica | Preliminar |
| **Qué** | Componentes, interfaces y políticas | Componentes, interfaces y políticas | Componentes, interfaces y políticas | Componentes y políticas | Componentes, algoritmos y protocolos |
| **Como** | Semi formal | Formal o semi formal | Semi formal | Semi formal o informal | Formal o semi formal |

Tabla 1 Tipos de arquitectura de referencia

La totalidad de las arquitecturas de referencia encontradas y detalladas a continuación se clasifican dentro del tipo 3. Las arquitecturas de referencia de tipo 3 contienen las siguientes principales características:

- El objetivo principal es facilitar y guiar a los arquitectos en los diseños de nuevas soluciones. Esto demuestra que todavía el área no tiene la madurez suficiente para lograr el objetivo de estandarización.
- El contexto es lo suficientemente genérico para aplicarlo en múltiples organizaciones en búsqueda de los principios de diseño arquitectónico. La audiencia principal son los arquitectos con la responsabilidad suficiente para tomar las decisiones de implementación de soluciones de big data dentro de su organización.
- Las arquitecturas se basan en soluciones existentes utilizadas y probadas por la industria.
- Los componentes, interfaces y políticas de la arquitectura se especifican a través de algún lenguaje semi-formal como diagramas.

### 2.3.1 Arquitectura de Maier

**Maier** [23] en su tesis de maestría describe una arquitectura de referencia con el propósito de facilitar y guiar la creación de soluciones de big data para múltiples organizaciones. La arquitectura se especifica a través de diagramas de UML bajo la arquitectura de vistas 4+1 de Krutchen [28]. A continuación se resumen las principales capas de la aplicación.

1. **Adquisición de datos**: Los datos pueden ser extraídos de un origen de datos dinámico, en el caso de un flujo de datos, o estático con datos en reposo. La extracción puede incluir la necesidad de filtrar los datos, que pueden ser estructurados, semi-estructurados o no estructurados.
2. **Extracción de información**: Para lograr generar información es necesario estructurar los datos con metadatos a través del reconocimiento y clasificación de entidades, así como sus relaciones. Se pueden utilizar ontologías para facilitar el entendimiento y extracción de los modelos propios de los datos.
3. **Gestión de calidad de datos**: La limpieza de datos permite la corrección de errores u omisión en los datos como pueden ser atributos faltantes, inconsistentes o duplicados.
4. **Integración de datos**: La integración permite armonizar los datos de diferentes orígenes a través de un esquema común para unificar las posibles consultas.
5. **Análisis de datos**: El análisis permite derivar significado e interpretación de los datos a través de la agregación de la información sobre diferentes dimensiones.
6. **Distribución de datos**: El objetivo de la distribución de datos es hacer disponible los resultados del análisis a través de interfaces de usuario o aplicación para los usuarios finales.

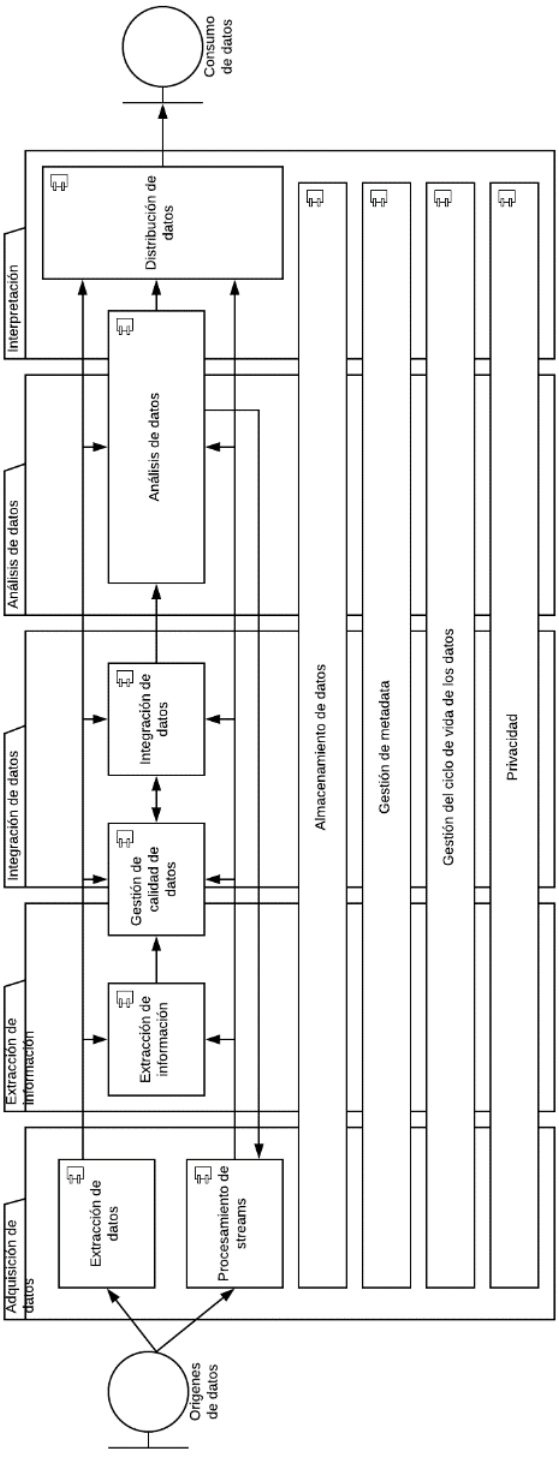


Figura 6 Arquitectura de referencia de Maier [23]

### 2.3.2 Arquitectura de Pääkkönen

Arquitectura de A su vez **Pääkkönen** [25] propone una arquitectura de referencia con el objetivo de facilitar el diseño y elección de tecnologías para su implementación. En la Figura 7 se muestra el diseño de alto nivel de dicha arquitectura que se resume a continuación.

El origen de datos se define en dos dimensiones: movilidad y estructura. La movilidad refiere al dinamismo y velocidad de los datos, mientras que la estructura define la información y metadata disponible para modelar los datos. La extracción puede incluir la transferencia, carga y compresión de los datos como forma de preprocesamiento. Los datos pueden ser combinados y limpiados para mejorar su calidad, así como replicados para su distribución. El proceso de extracción de información logra incorporar una mayor estructura a los datos más desestructurados. Los datos pueden ser analizados con procesamiento por lotes o en tiempo real y transformados para ser visualizados. La visualización puede ser realizada a través de dashboards o herramientas de reportes para el usuario final.

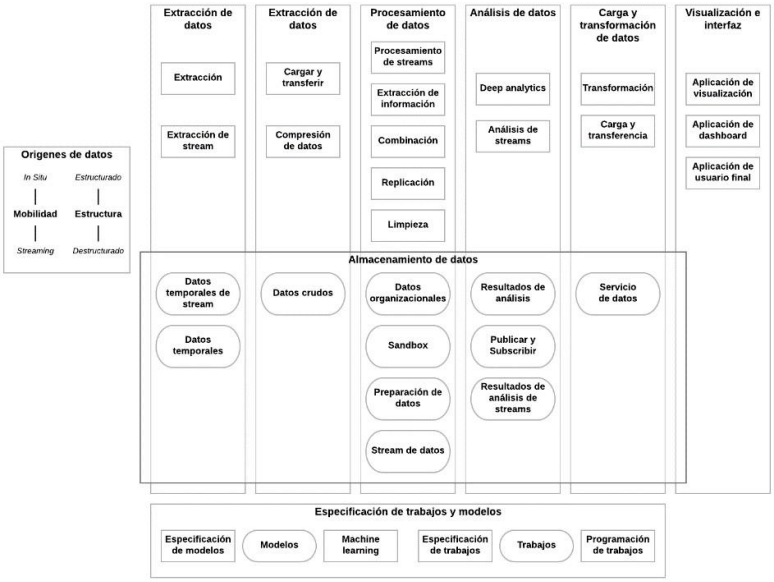


Figura 7 Arquitectura de referencia de Pääkkönen [25]

### 2.3.3 Arquitectura de Geerdink

**Geerdink** [24] detalla una arquitectura de referencia basada en dos patrones principales: layers y pipes & filters [29]. Los artefactos de la arquitectura (Figura 8) son creados iterando sobre el paradigma de diseño planteado por Hevner [30]. La arquitectura se compone de tres capas: negocio, aplicación y tecnológica. Se puede apreciar el uso de pipes & filters en la capa de negocio con la secuencia de importación, procesamiento, análisis y decisión de los datos.

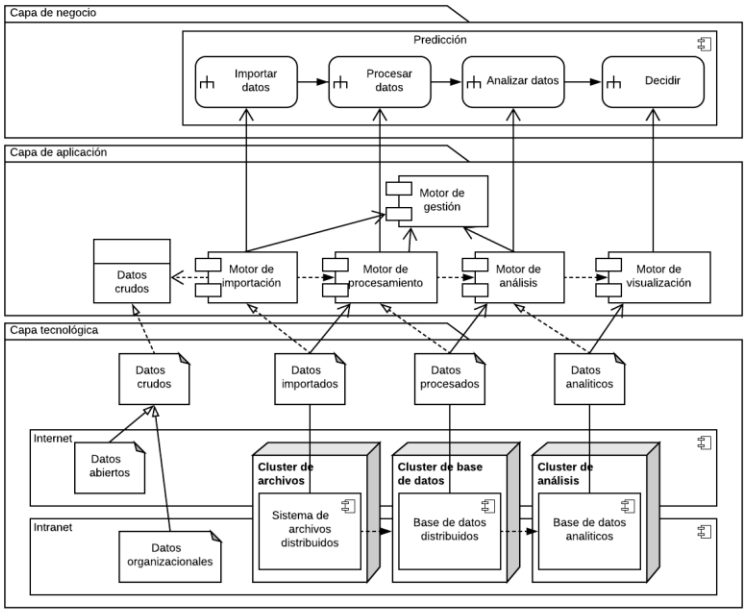


Figura 8 Arquitectura de referencia de Geerdink [24]

### 2.3.4 Arquitectura de Viana

**Viana** [26] focaliza su estudio en una arquitectura de referencia (Figura 9) para archivar, preservar y recuperar big data en el largo plazo. Existen otros estudios [31][32][33] sobre algunas capacidades puntuales de archivado, preservación y recuperación para datos estructurados y no estructurados.

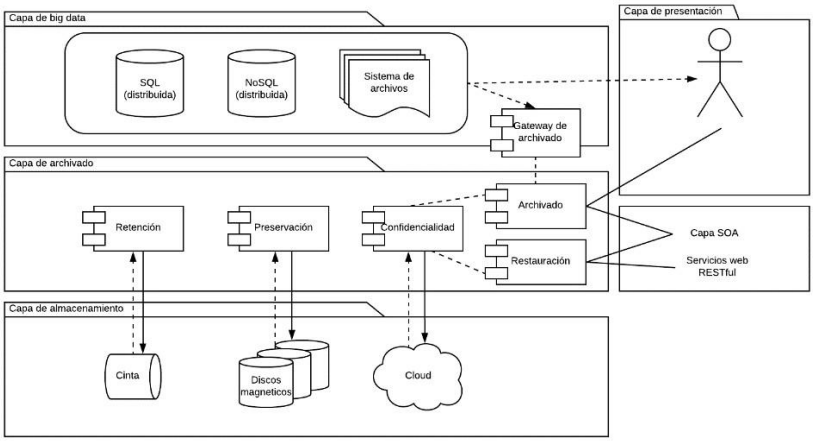


Figura 9 Arquitectura de referencia de Viana [26]

# 3 Tácticas arquitectónicas

Las soluciones de big data presentan varios desafíos para los arquitectos de software. En particular, la necesidad de distribuir las arquitecturas de software para cumplir las características propias de big data (volumen, variedad y velocidad) implica resolver aspectos de almacenamiento, procesamiento y visualización de datos. El arquitecto de software debe tomar decisiones de diseño que logren satisfacer los atributos de calidad propios de los escenarios de uso del sistema. Una buena forma de capturar estas decisiones de diseño es a través de las tácticas y patrones de arquitectura.

Una táctica de arquitectura [34] es una forma de controlar la respuesta del estímulo para satisfacer un atributo de calidad a través de decisiones de diseño de arquitectura. Especifica como los parámetros del modelo del atributo de calidad pueden ser controlados para conseguir una respuesta deseada frente al estímulo del artefacto. A continuación, describimos las tácticas de arquitectura de software principales identificadas en el diseño de las arquitecturas anteriormente descriptas.

La descripción de las tácticas se presenta con la siguiente estructura general.

- **Atributo de calidad**: Las tácticas arquitectónicas se agrupan por atributo de calidad que resuelven. Para un mejor entendimiento de cada atributo de calidad se describen sus escenarios de uso. Los escenarios se componen de un estímulo que ingresa al sistema para producir una respuesta esperada medible bajo ciertos contextos de uso. La medición de la respuesta proporciona el grado de satisfacción de dicho atributo de calidad.
- **Escenarios de uso general en big data**: De la evaluación de las arquitecturas de software se extrajeron escenarios de uso propios de big data. Cada escenario de uso general de big data tiene asociado la descripción de las tácticas arquitectónicas que buscan satisfacerlo.
- **Descripciones tácticas arquitectónicas**: La descripción contiene la explicación de la táctica arquitectónica con sus referencias principales encontradas en la literatura.

## 3.1 Escalabilidad

La escalabilidad describe como un sistema responde adecuadamente frente al incremento de carga. Un sistema escalable debería poder ajustar sus recursos para cubrir los cambios de demanda en su uso [34] como se describe en la Tabla 2.

| | |
|---|---|
| **Estímulo** | Incremento en la carga (demanda) en un recurso como almacenamiento, procesamiento o comunicación del sistema. |
| **Entorno** | Incremento en la carga es transitorio<br>Incremento en la carga es permanente |
| **Respuesta** | El sistema provee nuevos recursos para cubrir la demanda |
| **Medida de la respuesta** | Tiempo necesario para proveer nuevos recursos ajustados a la nueva demanda<br>Costo de nuevos recursos respecto al valor generado por la nueva demanda |

Tabla 2 Escenario de uso general de escalabilidad

Existen dos formas tradicionales de escalar los sistemas: horizontal y vertical. Escalar verticalmente implica incrementar los recursos sobre un único nodo computacional, mientras que escalar horizontalmente implica hacerlo sobre múltiples nodos. El scale cube [35] describe tres dimensiones diferentes de escalabilidad horizontal expresados en tres ejes de descomposición y replicación de los sistemas. El eje X del cubo describe como lograr escalar el procesamiento, a través de la replicación, redundancia y paralelización de los datos, sobre múltiples nodos bajo esquemas de balance de carga. El eje Y aplica una descomposición funcional del sistema en servicios con responsabilidades similares, como suelen promoverse en arquitecturas de microservicios. El eje Z propone dividir los datos en distintas particiones, lo que usualmente se conoce en base de datos como shards. Todos estos ejes describen diferentes dimensiones que se pueden combinar para escalar horizontalmente los sistemas.

### 3.1.1 Escenarios de uso general de escalabilidad en big data

Los dos escenarios principales en big data para la escalabilidad horizontal incluyen poder distribuir el almacenamiento y procesamiento de grandes volúmenes de datos en una variedad de formatos como se muestra en la Tabla 3.

| Escenario de uso | Táctica |
|---|---|
| Gestionar grandes volúmenes de datos | *Particionamiento de datos* |
| Procesar grandes volúmenes de datos | *Procesamiento por lotes*<br>*Entornos shared-nothing* |

Tabla 3 Tácticas de escalabilidad

### 3.1.2 Particionamiento de datos

El objetivo del particionamiento es distribuir los datos y sus consultas en forma equitativa entre los diferentes nodos del sistema. Esto genera el beneficio de poder escalar y distribuir la carga del sistema entre las diferentes particiones. Si no se logra una distribución equitativa pueden existir nodos, conocidos como "hot spots", con una mayor carga operativa perdiendo así los beneficios del particionamiento. De aquí

radica la importancia en la elección del esquema de particionamiento para lograr la mejor distribución de los datos. A continuación, se resumen dos esquemas utilizados para el particionamiento y la elección de las claves para identificar los elementos y realizar la distribución de los datos.

1. **Particionamiento por rangos de clave:** Para cada partición se generan rangos de claves ordenadas que facilitan la búsqueda de los datos al realizar las consultas. Sin embargo, son propensas a generar "hot spots"[1] si la aplicación accede regularmente a ciertos rangos consecutivos.
2. **Particionamiento por hash:** La distribución de los datos se realiza a través de una función de hash. Este método elimina el ordenamiento secuencial de las claves haciendo menos eficiente la búsqueda ordenada, pero asegura una mejor distribución equitativa de las particiones.

La evolución en los datos del sistema puede generar cambios en las particiones debido al incremento del volumen de datos en el tiempo. Esto implica que se deben volver a balancear las particiones para acomodar la nueva distribución de los datos.

### 3.1.3 Entorno share-nothing

Los entornos shared-nothing [36] permiten tener nodos independientes y auto-suficientes, evitando tener un único punto de contención del sistema y excesiva sincronización entre los nodos.

La clasificación de los entornos distribuidos en términos de su arquitectura puede realizarse en diferentes niveles. La clasificación de Stonebraker [36] permite distinguir tres grandes aproximaciones (Figura 10) shared-everything, shared-disk y shared-nothing.

[1] Cuando las particiones no son uniformemente distribuidas pueden generarse particiones con mayor demanda denominadas “hot spots”.

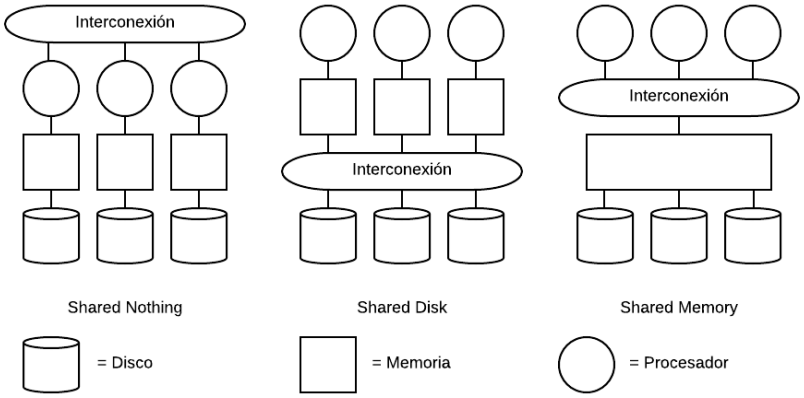


Figura 10 Entornos sistemas distribuidos

Los entornos shared-everything comparten todos los recursos de procesamiento, memoria y almacenamiento. Esto facilita la comunicación entre los componentes, pero a su vez genera un alto acoplamiento que dificulta la escalabilidad del sistema. Los sistemas shared-disk comparten el recurso de almacenamiento el cual se vuelve un único punto de contención. Sin embargo, estos esquemas se vuelven prohibitivos bajo sistemas de big data que necesitan escalar los datos y su procesamiento. Los entornos shared-nothing se vuelven útiles en estos contextos donde se necesita una alta escalabilidad y disponibilidad de los datos. En estos casos es necesario una red de alta velocidad para interconectar los diferentes nodos para conseguir las latencias esperadas.

La construcción de un sistema shared-nothing se puede dividir en los siguientes pasos [37]:

1. **Particionar los nodos:** Esto implica generar un particionamiento del sistema en nodos independientes y autosuficientes. Las particiones pueden ser a nivel de datos o funciones del sistema.
2. **Distribución de los nodos:** Una vez generada las particiones se deben distribuir las instancias de datos y procesamiento dentro de cada nodo.
3. **Configuración del balanceo de carga:** Dada la elección de la partición se debe configurar la distribución y balanceo de la carga del sistema entre sus diferentes nodos.

Este proceso hay que repetirlo en casos que se quiera volver a particionar el sistema de forma diferente.

### 3.1.4 Procesamiento por lotes

El procesamiento por lotes permite la ingesta de un gran volumen de datos a través de un proceso que generalmente se programa para ejecutarse periódicamente. Un

sistema de procesamiento por lotes permite procesar lotes de datos a través de una serie de procesos o trabajos sin necesidad de intervención manual. En general los procesos son ejecutados regularmente y pueden tardar un tiempo considerable (lo que los distingue del procesamiento en tiempo real). La principal medida de eficiencia del procesamiento por lotes es el tamaño (o volumen) de datos que se puede procesar bajo cierto tiempo determinado.

Un modelo de programación muy utilizado en estos esquemas es el MapReduce [22]. El proceso se puede visualizar como funciones de transformación de entradas en salidas bajo dos procesos principales de mapeo y reducción.

**Mapeo**: El mapeo es realizado una vez por cada registro de entrada después de extraer su clave y valor a ser procesado. Por cada entrada se genera un número de pares nuevos de clave y valor a ser ordenados y reducidos. Este proceso no guarda el estado de procesamiento de entradas anteriores por lo cual se puede realizar en paralelo y de forma independiente. El proceso de mapeo puede ser resumido bajo la transformación Map(k1, v1) => List(k2, v2).

**Reducción**: La nueva lista de clave y valor producidos en el proceso de mapeo son agrupados bajo las mismas claves para ser reducidos. Cada proceso de reducción procesa todos los valores asociados a una clave para generar el resultado final. Este proceso en su conjunto genera una nueva lista de valores reducidos por cada una de las claves procesadas. El proceso de reducción puede ser resumido bajo la transformación Reduce(k2, list(v2)) -> list(k2, v3).

Estos dos procesos permiten procesar grandes volúmenes de datos en forma distribuida y paralela como se describe en el proceso más detallado de la Figura 11.

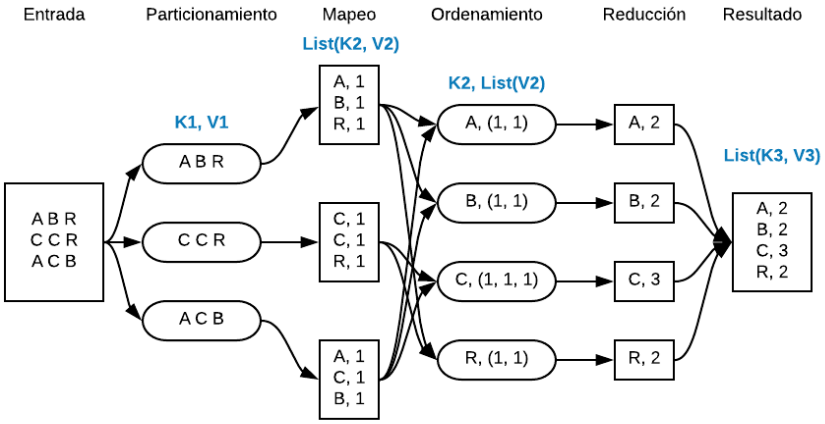


Figura 11 Proceso MapReduce

1. **Particionar:** Se leen un conjunto de entradas que se dividen en registros con claves a ser procesados por cada valor.
2. **Mapear:** Se utiliza el proceso de mapeo para procesar la clave y valor de cada registro.
3. **Ordenar:** Se agrupan ordenadamente los registros por clave para ser procesados por cada reductor.
4. **Reducir:** Se utiliza el proceso de reducción para iterar y procesar cada registro ordenado por su clave para obtener el resultado final.

## 3.2 Eficiencia

La eficiencia es la capacidad del sistema para responder bajo tiempos específicos. Caracterizar los eventos que suceden dentro del sistema y sus tiempos de respuesta es la esencia principal en la discusión de este atributo de calidad [34] como lo muestra la Tabla 4.

| | |
|---|---|
| **Estímulo** | Arribo de eventos o solicitudes esporádicos al sistema<br>Arribo de eventos o solicitudes periódicos al sistema<br>Arribo de eventos o solicitudes estocásticos al sistema |
| **Entorno** | Carga intensiva de lecturas/escrituras<br>Carga intensiva de procesamiento<br>Clientes centralizados o distribuidos geográficamente |
| **Respuesta** | Se procesa cada solicitud<br>Cambio en el nivel de respuesta del servicio |
| **Medida de la respuesta** | Latencia promedio<br>Peor caso de latencia<br>Varianza de latencia<br>Ratio de demanda pico<br>Promedio de demanda |

Tabla 4 Escenario de uso general de eficiencia

### 3.2.1 Escenarios de uso general de eficiencia en big data

Podemos analizar las latencias en soluciones de big data del punto de vista de la gestión de sus datos, así como de los tiempos utilizados en su procesamiento y análisis. La Tabla 5 describe las diferentes tácticas de eficiencia de acuerdo con el tipo de respuesta que queremos lograr.

| Escenario de uso | Táctica |
|---|---|
| Alta frecuencia de generación y lectura de datos | *Consistencia eventual* |
| Alta frecuencia en el análisis y publicación de resultados. | *Procesamiento en tiempo real* |

Tabla 5 Tácticas de eficiencia

### 3.2.2 Consistencia eventual

En sistemas distribuidos, bajo múltiples nodos y particiones, el teorema del CAP nos sugiere que debemos balancear entre disponibilidad y consistencia [38]. Para lograr sistemas altamente disponibles debemos entonces relajar su nivel de consistencia. La introducción del concepto de consistencia eventual [18] permite sincronizar los datos de las particiones de forma eventual, incluso frente a fallas que puedan ocurrir entre las particiones. La sincronización de cambios entre réplicas no es instantánea y dependiendo del nivel de consistencia que se quiera lograr existen diferentes modelos.

Un modelo de consistencia es un contrato entre el almacenamiento de datos y el proceso que lo consume. Normalmente cuando un proceso consulta un dato, la operación espera obtener la última versión escrita del elemento. En caso de existir múltiples réplicas de los datos puede generarse una inconsistencia en las versiones replicadas en el tiempo.

Existen distintos modelos o niveles de consistencia de acuerdo con el balance que se quiera lograr entre la latencia de las operaciones de lectura/escritura y la consistencia de los datos replicados. A continuación, se describen los modelos principales (Figura 12**Error! Reference source not found.**) [39].

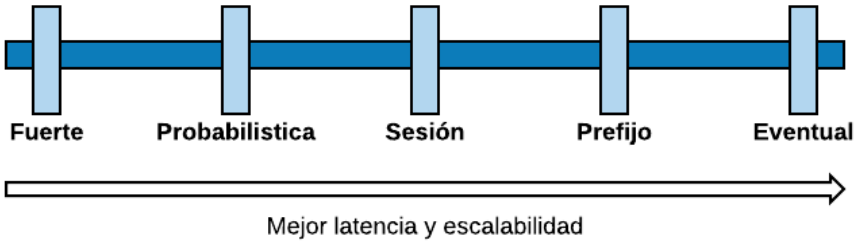


Figura 12 Modelos de consistencia

- **Consistencia fuerte:** Este modelo asegura la linealidad de la información. Las lecturas garantizan que las escrituras sean visibles una vez que son confirmadas en consenso por la mayoría de las réplicas.
- **Consistencia probabilística:** Este nivel permite calcular lo eventual y consistente de los datos con un modelo probabilístico. Se utilizan dos métricas para predecir el nivel de consistencia eventual. Por un lado, el tiempo de consistencia se calcula con la probabilidad de leer un elemento después de un tiempo t de ser escrito. Esto permite contestar el grado de eventualidad del modelo. Por otro lado, la probabilidad de consultar una versión anterior K veces menor a la última versión escrita hasta el momento, permite estimar la consistencia del modelo. La combinación de estas dos métricas permite generar un estimativo del tiempo total necesario para replicar cada versión del elemento escrito.
- **Consistencia por sesión:** Este modelo asegura una consistencia local por sesión de usuario y no global como las dos anteriores. Este nivel asegura lecturas monotónicas, donde cada lectura obtendrá versiones del elemento más recientes a las lecturas anteriores. Las escrituras mantendrán un orden consecutivo dentro de la misma sesión de usuario, lo que se conoce como escrituras monotónicas. Finalmente, las últimas escrituras siempre serán obtenidas en posteriores lecturas dentro de la misma sesión, pero no es garantizado entre diferentes sesiones.
- **Consistencia por prefijo:** La consistencia por prefijo asegura que las lecturas son obtenidas en el mismo orden que ocurren las escrituras. Si las escrituras se generan en el orden A, B, C, entonces un cliente debe seguir viendo ese mismo orden en sus lecturas.

- **Consistencia eventual:** Es la forma más débil de consistencia, pero se obtienen las mejores latencias de escrituras y lecturas. Se garantiza qué en la ausencia de más escrituras, las réplicas convergen eventualmente en el tiempo, pero no se asegura el orden que el cliente obtiene los valores escritos.

### 3.2.3 Procesamiento en tiempo real

Mientras que los procesamientos por lotes necesitan de todos los registros de entrada para generar los resultados, existe el concepto de streams para lograr el procesamiento en tiempo real. Los streams son un flujo de datos que se obtienen incrementalmente en el tiempo y pueden no tener un final determinado. En el contexto del procesamiento de streams, a un registro de entrada se le conoce como un evento. El evento es un objeto inmutable que contiene los detalles de un suceso en el tiempo. El evento es generado por un productor y puede ser procesado por múltiples consumidores. Generalmente los eventos relacionados se pueden agrupar en tópicos donde los consumidores pueden subscribirse para consumirlos.

Para sincronizar el consumo de los distintos subscriptores se debe utilizar alguna estructura de almacenamiento de los eventos en el tiempo con sus correspondientes notificaciones. Este tipo de requerimientos pueden ser implementados con sistemas de mensajería, donde el consumidor genera un evento en forma de mensaje y este es notificado a los diferentes consumidores. Algunos desafíos en estos sistemas son la sincronización de procesamiento entre el productor y consumidor, y el manejo de la pérdida de mensajes frente a caídas de los productores y/o consumidores. Una forma de resolver estos casos es a través de colas o buffers que permiten agrupar los eventos producidos en diferentes tópicos. Estos tópicos permiten agrupar, persistir y retener los mensajes para recuperar posibles caídas y retrasos de los subscriptores en el consumo de sus mensajes. Incluso la retención indefinida de mensajes suele utilizarse para poder agregar fácilmente futuros subscriptores que quieran procesar la totalidad de los eventos dentro de un tópico en particular.

## 3.3 Modificabilidad

La modificabilidad es la facilidad del sistema para analizar, implementar, probar y desplegar modificaciones sin introducir defectos o degradar la calidad del producto actual [40]. Para esto se debe entender el origen, frecuencia, complejidad y costo del cambio como describe la Tabla 6.

| | |
|---|---|
| **Estímulo** | Agregar/Eliminar/Variar<br>Funcionalidad<br>Formato/Estructura de datos<br>Capacidad o tecnología de almacenamiento/procesamiento |
| **Entorno** | Tiempo de diseño<br>Tiempo de compilación<br>Tiempo de ejecución |
| **Respuesta** | Localizar lugares en la arquitectura para realizar cambios<br>Realizar modificaciones sin afectar otras partes del sistema<br>Probar modificaciones<br>Desplegar modificaciones |
| **Medida de la respuesta** | Dificultad del cambio en términos de<br>Componentes afectados<br>Esfuerzo/Tiempo/Costo |

Tabla 6 Escenario de uso general de modificabilidad

### 3.3.1 Escenarios de uso general de modificabilidad en big data

La característica de variedad de big data implica gestionar una diversidad de formatos de datos. Por un lado, existen los datos estructurados que pueden ser descriptos bajo un esquema y estructura fija. Por otro lado, los datos semi-estructurados surgen cuando no existe una estructura rígida (como la Web) y generalmente combinan diferentes orígenes heterogéneos de datos. Los datos semi-estructurados son caracterizados por una estructura flexible y autodescriptiva. Por último, encontramos los datos no estructurados, como generalmente son los archivos de multimedia (texto, video, imágenes), que no contienen ninguna estructura o esquema asociado. La alta diversidad de formatos genera el desafío de persistir y gestionar los datos bajo una misma solución.

| Escenario de uso | Táctica |
|---|---|
| Múltiples formatos de datos (estructurados, semi-estructurados y no estructurados) | *Persistencia políglota* |

Tabla 7 Tácticas de modificabilidad

### 3.3.2 Persistencia políglota

Las bases de datos relacionales permiten unificar e integrar los datos de varias aplicaciones en una única fuente de datos. En este escenario, la base de datos puede actuar como integradora de múltiples aplicaciones, usualmente desarrollada por diferentes equipos, bajo un almacenamiento común. Esto facilita la consistencia y transaccionalidad de las operaciones bajo las propiedades denominadas ACID [41].

- **Atómica**: Todas las operaciones de una transacción se realizan en forma atómica. Si sucede una falla en una de las operaciones se cancela la transacción en su conjunto. Esto permite asegurar la integridad de los datos.

- **Consistente**: Existe una consistencia lineal donde cada escritura queda disponible inmediatamente para posteriores lecturas y consultas.
- **Aislada**: El aislamiento transaccional permite asegurar que las operaciones concurrentes sean sincronizadas correctamente.
- **Durable**: Cuando se completa una transacción esta debe persistir incluso ante posibles fallas y caídas del sistema.

Sin embargo, la variedad de formatos implica complejizar el diseño del modelo común para acomodar las diferentes estructuras y esquemas heterogéneos de datos. Si una de las aplicaciones quiere realizar modificaciones al modelo, esto tiene un impacto directo sobre las otras aplicaciones que la utilizan. Además existe la dificultad de evolucionar y migrar los esquemas durante el ciclo de vida del producto [42]. La desventaja de los modelos con esquemas rígidos es que no tienen la flexibilidad de aplicar migraciones diferidas [43] para evitar interrupciones [44] en el servicio de la aplicación. Por otro lado, la necesidad de manejar un gran volumen de datos es limitado en contextos ACID donde solo se puede escalar verticalmente (sobre un único nodo). Esto se debe a que se prioriza la fuerte consistencia sobre la disponibilidad bajo múltiples particiones que permitan escalar horizontalmente.

Para resolver estos desafíos surgen las bases de datos NoSQL con las propiedades denominadas BASE [45] (Basic availability, Soft-state, Eventual consistency). Estas propiedades priorizan la disponibilidad y escalabilidad a través de la redundancia y particionamiento en múltiples réplicas. Esto implica que las réplicas no siempre serán consistentes, pero podrán escalar horizontalmente y sincronizarán eventualmente. Existe una variedad de base de datos NoSQL que se pueden categorizar de acuerdo a su modelo de datos [46] como resume la Tabla 8.

| Modelo de datos | Descripción | Tipo de estructura |
|---|---|---|
| Clave-valor | Consiste en un conjunto de pares de clave-valor únicos. Su simple estructura permite operaciones de creación, lectura, borrado y actualización. No impone ningún esquema de datos para sus valores, lo que restringe las consultas unicamente por clave. | No estructurado |
| Documental | Consiste en un almacenamiento clave-valor pero restringe los valores a algún formato semi-estructurado como documentos JSON. Esta restricción de estructura agrega una mayor capacidad de consulta sobre el propio documento. | Semi-estructurado |
| Familia de columnas | Consiste en familias de columnas ordenadas por fila. Cada columna tiene su propia clave de acceso así como cada una de sus filas. La estructura está optimizada para almacenar grandes cantidades de columnas por fila. | Estructurado |
| Grafos | Permite almacenar entidades con sus relaciones. Las entidades son nodos o vértices de un grafo con propiedades. Las relaciones se representan como aristas con propiedades que conectan los diferentes vértices. La estructura del grafo está optimizada para poder navegar las diferentes relaciones y sus nodos asociados. | Semi-estructurado |

Tabla 8 Tipos de base de datos NoSQL

La persistencia políglota [47] propone utilizar múltiples modelos de datos, bajo una misma solución, de acuerdo a cada necesidad particular de persistencia. Esto deriva del concepto de utilizar la herramienta adecuada al problema a resolver. Esta diversidad de modelos de datos genera un desafío adicional al practicante que debe poder elegir la mejor opción de acuerdo con su contexto de uso. Existe un interés en la comunidad [41] [46] para evaluar la elección del modelo de datos adecuado según el contexto de uso. Un ejemplo de esto es la base de conocimiento del SEI [48] para la evaluación de base de datos distribuidas denominado QuABaseBD[2] (Quality Attributes at Scale Knowledge Base). Por otro lado, existen base de datos multi-modelos [49] que agrupan los beneficios de diferentes modelos de datos bajo una misma base de datos.

## 3.4 Disponibilidad

La disponibilidad refiere a la habilidad del sistema para enmascarar o reparar faltas que puedan provocar fallas e inconvenientes a los usuarios finales [34]. En sistemas distribuidos, la probabilidad de fallas en el sistema se incrementa, con el aumento en la multiplicidad e interconexión de sus nodos. Los sistemas deben manejar estos

[2] https://quabase.sei.cmu.edu

errores para mitigar las fallas que puedan suceder en tiempo de ejecución como lo muestra la Tabla 9.

| | |
|---|---|
| **Estímulo** | Una falla sucede en un recurso del sistema.<br>El recurso puede ser un nodo de procesamiento, comunicación o almacenamiento. |
| **Entorno** | Ocurren una o multiples fallas simultaneamente |
| **Respuesta** | El sistema continua procesando solicitudes sin degradar su nivel de servicio.<br>El sistema procesa solicitudes pero degradando su nivel de servicio. |
| **Medida de la respuesta** | Tiempo para restaurar el recurso ante una falla temporaria<br>Tiempo para sustituir el recurso ante una falla permanente<br>Cantidad de fallas que pueden ser enmascaradas o toleradas. |

Tabla 9 Escenario de uso general de disponibilidad

### 3.4.1 Escenarios de uso general de disponibilidad en big data

En los sistemas distribuidos la disponibilidad típicamente se logra a través de la redundancia de sus componentes. La replicación permite mantener varias copias de los componentes y datos en diferentes nodos (potencialmente distribuidos geográficamente). Ante la falla o caída de algún nodo, la replicación permite la redundancia para lograr servir los datos o componentes desde los nodos operativos restantes.

| Escenario de uso | Táctica |
|---|---|
| Recuperar ante fallas en el almacenamiento, procesamiento y comunicación de la plataforma | *Write ahead logs*<br>*Semánticas de llamadas remotas* |
| Tolerar fallas en el almacenamiento, procesamiento y comunicación de la plataforma | *Replicación de datos*<br>*Replicación funcional de procesamiento* |

Tabla 10 Tácticas de disponibilidad

### 3.4.2 Write-Ahead Log

Una variedad de bases de datos ha utilizado el mecanismo de write-ahead logs como fuente para recuperarse frente a fallas durante la caída de sus transacciones. Un log [50] es una secuencia inmutable de registros ordenadas temporalmente donde solo se pueden agregar nuevos datos al final de la secuencia. Los nuevos registros se añaden al final de la secuencia y el contenido es leído desde el comienzo en forma secuencial hasta el final del log. Cada entrada del log contiene un número único secuencial para lograr un ordenamiento de los registros como se muestra en la Figura 13.

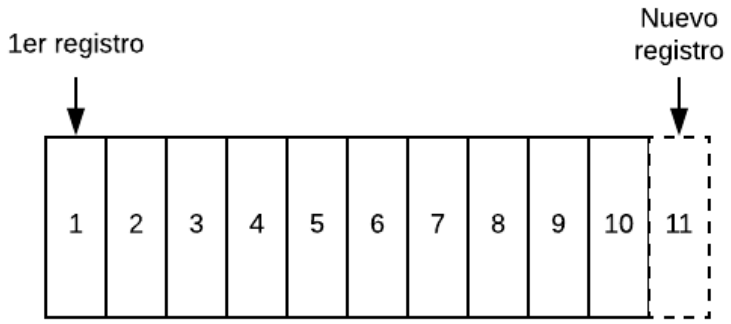


Figura 13 Estructura de un log

Esta ordenación asegura procesamientos determinísticos incluso bajo esquemas de alta distribución y replicación. Al partir de un mismo estado y procesar los registros bajo la misma secuencia de eventos se asegura un mismo resultado cada vez. Esto facilita la distribución del procesamiento de un gran volumen de datos, permitiendo utilizar un esquema de productor/suscriptor para analizar el log. Los productores generan el log en forma secuencial a partir de los eventos o datos generados en el sistema. Esto facilita que cada subscriptor pueda procesar los registros a su propio ritmo. Cada subscriptor utiliza el índice dentro de la secuencia del log para identificar su etapa de procesamiento. Ante posibles fallas y bajo la propiedad de inmutabilidad del log, los subscriptores pueden reiniciar su procesamiento desde el lugar donde se produjo la falla.

Este esquema permite guardar todos los eventos del sistema y reproducir el estado del sistema en cada momento. Si el procesamiento se realiza en tiempo real, el log se transforma en un flujo (stream) de datos que se actualiza en tiempo real con cada nuevo registro producido. Este mecanismo utiliza una única estructura de datos simple, escalable y sincronizable entre múltiples productores y subscriptores. Agregar otra forma de análisis de los eventos implica agregar otro subscriptor con la nueva lógica de procesamiento.

### 3.4.3 Semánticas de llamadas remotas

En sistemas distribuidos también hay que lidiar con fallas que puedan producirse durante las llamadas remotas. Existen diferentes semánticas de llamadas [51] entre componentes del sistema para enviarse mensajes. Idealmente queremos garantizar el envío único del mensaje a través de cada llamada. Sin embargo, esto no es sencillo de conseguir en sistemas distribuidos donde pueden existir diferentes fallas durante la comunicación y envío de mensajes. Es por esto que existen diferentes semánticas de llamadas como se describen a continuación.

- **At most once:** En este modelo el envío del mensaje se realiza una única vez. Esto implica que si existen fallos puede que no se envíe el mensaje correctamente. Este modelo se utiliza cuando la pérdida ocasional de mensajes no es un problema.
- **At least once:** Este modelo asegura que el receptor recibe al menos una vez el mensaje. Para garantizar la comunicación, el emisor reintenta el envío del mensaje hasta que el receptor confirma su correcta recepción. Esta aproximación puede ser utilizada cuando la operación a realizar es idempotente, ya que no podemos asegurar la unicidad de la llamada por reintentos.
- **Exactly once:** Este es el caso ideal donde se consigue enviar el mensaje exactamente una única vez. Sin embargo, esto no es posible de lograr en sistemas distribuidos donde pueden existir diferentes tipos de fallas durante la comunicación. El resultado final puede ser simulado a través de aspectos transaccionales e idempotentes de las operaciones bajo esquemas de at least once.

### 3.4.4 Replicación de datos y funcional

La replicación es una forma de lograr la redundancia de los diferentes componentes de una arquitectura para enmascarar sus posibles fallas de sistema. La replicación implica tener varias copias de los mismos datos o funciones distribuidos en múltiples nodos denominadas réplicas. Existen diferentes beneficios de la replicación como pueden ser:

- Reducir la latencia manteniendo las copias cercanas a los consumidores finales bajo esquemas de distribución geográficamente.
- Incrementar la disponibilidad permitiendo al sistema seguir funcionando incluso ante fallas que puedan suceder en un número determinado de réplicas.
- Aumentar el rendimiento del sistema al poder escalar la cantidad de nodos que sirven datos o realicen procesamiento.

La dificultad de la replicación radica en los cambios que se deben propagar para mantener las réplicas sincronizadas. Existen diferentes algoritmos de replicación en la literatura para lograr este objetivo. A continuación se describen los tres principales [52].

1. **Replicación basada en un líder (Leader-based replication):** En este caso una de las réplicas es designada como el líder (también conocido como master o primario). Todas las escrituras son enviadas al líder para ser persistidas. El resto de las réplicas son seguidores y reciben los cambios a sincronizar a través del líder. Cuando un cliente quiere leer los datos lo puede hacer desde

el líder, o alguno de los seguidores, en cambio las escrituras son siempre realizadas a través del líder.

2. **Replicación multi líder (Multi-leader replication):** Este modelo es una extensión del anterior, donde existen múltiples líderes que pueden recibir escrituras. Esto implica que los líderes deben poder sincronizarse con los otros líderes del sistema. Existen diferentes topologías para realizar estas sincronizaciones como puede ser el cirular, estrella o entre todos los líderes. Esto es de utilidad en contextos de centros de datos distribuidos geográficamente, donde cada centro tiene su propio líder además de sus seguidores. Esta aproximación agrega la complejidad de resolver los conflictos de escritura entre líderes. Existen diferentes formas de resolución de conflictos, aunque ninguna óptima ya que depende del dominio de aplicación. Esto hace que muchas herramientas permitan implementar código particular para resolver dichas situaciones particulares.
3. **Replicación sin líder (Leaderless replication):** En este modelo no existe el concepto de líderes y las escrituras y lecturas se realizan a un número determinado de nodos en forma simultánea. Para asegurarse la consistencia de datos el número de escrituras y lecturas deben solaparse para asegurar que las lecturas se realizan sobre al menos un nodo con las últimas versiones de los datos.

# 4 Conclusiones

El presente catálogo busca describir los escenarios de uso y atributos de calidad encontrados bajo los contextos de big data y las tácticas de arquitectura que guíen su resolución. Del análisis de las descripciones de las arquitecturas identificadas en el mapeo sistemático se pueden elaborar escenarios de uso con problemáticas comunes. Muchos de los desafíos son similares a los encontrados en sistemas distribuidos [53] pero a mayor escala determinados por las 3Vs (volumen, variedad y velocidad) [54]. Esto tiene un impacto directo en las decisiones de diseño que un arquitecto de software debe contemplar frente a situaciones similares. Los sistemas de big data son inherentemente distribuidos, por lo que sus arquitecturas deben lidiar explícitamente con las fallas, latencias de comunicación, concurrencia, consistencia y replicación de sus múltiples nodos y componentes distribuidos. Al crecer y replicar el sistema a miles de nodos distribuidos geográficamente estos desafíos aumentan con la mayor probabilidad de fallas en el hardware y software [55].

Los atributos propios de los contextos de big data generan la necesidad de tener arquitecturas complejas de varias capas. Esto se puede observar en las arquitecturas de referencia que contienen una variedad de capas (cada una con sus propios desafíos tecnológicos) para lograr la persistencia, extracción, procesamiento, análisis, transformación y visualización de datos con características de big data. Por lo tanto, es crucial el entendimiento y diseño de cada componente para conseguir una solución integral de software. Esto implica muchas veces un gran desafío para el practicante que debe manejar los diferentes conceptos y tecnologías propias de cada capa.

Una buena forma de capturar estas decisiones de diseño es a través de las tácticas y patrones de arquitectura. Estos elementos son centrales para poder hacer un análisis acertado en las decisiones de diseño y lograr que tengan un impacto positivo sobre los requerimientos propios de big data. Es por esto que se entiende que las tácticas de arquitectura son esenciales para guiar al arquitecto en la toma de decisiones de diseño en la solución final. Esto se vuelve especialmente importante en contextos de big data donde el estudio identifica la complejidad de los requerimientos y la necesidad de utilizar tácticas particulares para su correcto diseño e implementación.

# 5 Referencias bibliográficas


[1] J. P. Russo, "Mapeo sistemático y evaluación de arquitecturas de software para contextos de big data," Universidad ORT Uruguay, 2018.

[2] J. P. Russo and M. Solari, "Estudio de Mapeo Sistemático sobre Arquitecturas de Software para Big Data," in *XX Ibero-American Conference on Software Engineering (CIbSE)*, 2017.

[3] R. Sumbaly, J. Kreps, and S. Shah, "The big data ecosystem at LinkedIn," in *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, 2013, p. 1125.

[4] J. Kreps, L. Corp, N. Narkhede, and J. Rao, "Kafka: a Distributed Messaging System for Log Processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[6] L. Wu *et al.*, "Avatara: OLAP for Web-scale Analytics Products," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1874–1877, 2012.

[7] "Using set cover algorithm to optimize query latency for a large scale distributed graph | LinkedIn Engineering." [Online]. Available: https://engineering.linkedin.com/real-time-distributed-graph/using-set-cover-algorithm-optimize-query-latency-large-scale-distributed. [Accessed: 14-Jul-2018].

[8] "Migrating to Espresso | LinkedIn Engineering." [Online]. Available: https://engineering.linkedin.com/blog/2017/08/migrating-from-oracle-to-espresso. [Accessed: 14-Jul-2018].

[9] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin, "Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 1147–1158.

[10] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy, "The unified logging infrastructure for data analytics at Twitter," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1771–1780, 2012.

[11] J. Lin and D. Ryaboy, "Scaling big data mining infrastructure," *ACM SIGKDD Explor. Newsl.*, vol. 14, no. 2, p. 6, Apr. 2013.

[12] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-Time Search at Twitter," in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 1360–1369.

[13] A. Thusoo *et al.*, "Data warehousing and analytics infrastructure at facebook," *Proc. 2010 ACM SIGMOD Int. Conf. Manag. data*, pp. 1013–1020, 2010.

[14] X. Amatriain, “Big & Personal: data and models behind Netflix recommendations,” in *Proceedings of the 2nd international workshop on big data, streams and heterogeneous source Mining: Algorithms, systems, programming models and applications*, 2013, pp. 1–6.

[15] NIST Group Big Data Public Working, “NIST Big Data Interoperability Framework: Volume 3, Use Cases and General Requirements,” *NIST Spec. Publ.*, 2015.

[16] N. Marz and J. Warren, *Big data : principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015.

[17] J. Kroß, A. Brunnert, C. Prehofer, T. A. Runkler, and H. Krcmar, “Stream processing on demand for lambda architectures,” in *European Workshop on Performance Engineering*, 2015, vol. 9272, pp. 243–257.

[18] P. Bailis and A. Ghodsi, “Eventual consistency today,” *Commun. ACM*, vol. 56, no. 5, p. 55, May 2013.

[19] M. Hausenblas and N. Bijnens, “Lambda Architecture,” 2015. [Online]. Available: http://lambda-architecture.net/. [Accessed: 29-Oct-2017].

[20] “Kappa Architecture - Where Every Thing Is A Stream.” [Online]. Available: http://milinda.pathirage.org/kappa-architecture.com/. [Accessed: 29-Oct-2017].

[21] C. E. Cuesta, M. A. Martínez-Prieto, and J. D. Fernández, “Towards an architecture for managing big semantic data in real-time,” in *European Conference on Software Architecture*, 2013, vol. 7957 LNCS, pp. 45–53.

[22] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[23] M. Maier, “Towards a Big Data Reference Architecture,” University of Eindhoven, 2013.

[24] B. Geerdink, “A reference architecture for big data solutions introducing a model to perform predictive analytics using big data technology,” in *8th International Conference for Internet Technology and Secured Transactions (ICITST-2013)*, 2013, pp. 71–76.

[25] P. Pääkkönen and D. Pakkala, “Reference Architecture and Classification of Technologies, Products and Services for Big Data Systems,” *Big Data Res.*, vol. 2, no. 4, pp. 166–186, Feb. 2015.

[26] P. Viana and L. Sato, “A proposal for a reference architecture for long-term archiving, preservation, and retrieval of big data,” in *Proceedings - 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014*, 2015, pp. 622–629.

[27] S. Angelov, P. Grefen, and D. Greefhorst, “A framework for analysis and design of software reference architectures,” *Inf. Softw. Technol.*, vol. 54, no. 4, pp. 417–

431, Apr. 2012.

[28] P. B. Kruchten, “The 4+1 View Model of architecture,” *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, 1995.

[29] P. Avgeriou and U. Zdun, “Architectural patterns revisited – a pattern language,” *10th Eur. Conf. Pattern Lang. Programs (EuroPlop 2005), Irsee*, 2005.

[30] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS Q.*, vol. 28, no. 1, pp. 75–105, 2004.

[31] D. Huhnlein, U. Korte, L. Langer, and A. Wiesmaier, “A Comprehensive Reference Architecture for Trustworthy Long-Term Archiving of Sensitive Data,” in *2009 3rd International Conference on New Technologies, Mobility and Security*, 2009, pp. 1–5.

[32] M. Butler, D. Reynolds, I. Dickinson, B. McBride, D. Grosvenor, and A. Seaborne, “Semantic Middleware for E-Discovery,” in *2009 IEEE International Conference on Semantic Computing*, 2009, pp. 275–280.

[33] EDRM, “The electronic discovery reference model.” [Online]. Available: http://www.edrm.net/resources/guides/edrm-framework-guides. [Accessed: 03-Mar-2018].

[34] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice. 3er ed*. Addison-Wesley, 2013.

[35] M. L. Abbott and M. T. Fisher, *The art of scalability : scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

[36] M. Stonebraker, “The Case for Shared Nothing,” *Database Eng.*, vol. 9, no. 1, pp. 4–9, 1986.

[37] T. Müseler, “A survey of Shared-Nothing Parallel Database Management Systems [Comparison between Teradata, Greenplum and Netezza implementations],” *IRCSE*, 2012.

[38] “You Can’t Sacrifice Partition Tolerance | codahale.com.” [Online]. Available: https://codahale.com/you-cant-sacrifice-partition-tolerance/. [Accessed: 19-Nov-2017].

[39] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, “Consistency-based service level agreements for cloud storage,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP ’13*, 2013, pp. 309–324.

[40] Organisation internationale de normalisation, *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. ISO/IEC 25010, 2011.

[41] G. Deepak, “A Critical Comparison of NOSQL Databases in the Context of Acid and Base,” *IRCSE*, 2016.

[42] T. Cerqueus, E. C. de Almeida, and S. Scherzinger, “Safely Managing Data Variety in Big Data Software Development,” in *2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering*, 2015, pp. 4–10.

[43] M. Klettke, U. Storl, M. Shenavai, and S. Scherzinger, “NoSQL schema evolution and big data migration at scale,” in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 2764–2774.

[44] I. Labs, T. Dumitras¸and, D. Dumitras¸and, and M. Hicks, “Evolving NoSQL Databases Without Downtime,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 166–176.

[45] D. Ganesh Chandra, “BASE analysis of NoSQL database,” *Futur. Gener. Comput. Syst.*, vol. 52, pp. 13–21, Nov. 2015.

[46] F. Gessert, W. Wingerath, S. Friedrich, and N. Ritter, “NoSQL database systems: a survey and decision guidance,” *Comput. Sci. - Res. Dev.*, vol. 32, no. 3–4, pp. 353–365, Jul. 2017.

[47] G. C. Deka, “NoSQL Polyglot Persistence,” in *Advances in Computers*, Elsevier, 2018, pp. 357–390.

[48] I. Gorton, J. Klein, and A. Nurgaliev, “Architecture Knowledge for Evaluating Scalable Databases,” in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, 2015, pp. 95–104.

[49] E. Płuciennik and K. Zgorzałek, “The Multi-model Databases – A Review,” Springer, Cham, 2017, pp. 141–152.

[50] J. Kreps, *I love logs : event data, stream processing, and data integration*. O’Reilly Media, 2015.

[51] A. D. Birrell and B. J. Nelson, “Implementing remote procedure calls,” *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[52] M. Kleppmann, *Designing data-intensive applications : the big ideas behind reliable, scalable, and maintainable systems*. O’Reilly Media, Inc., 2017.

[53] J. Pérez-Martínez and A. Sierra, “A Taxonomy of the Quality Attributes for Distributed Applications.” Informatica, 2002.

[54] D. Laney, “3D Data Management: Controlling Data Volume, Velocity and Variety,” *META Gr. Res. note*, vol. 6, no. 70, p. 1, 2001.

[55] K. V. Vishwanath and N. Nagappan, “Characterizing cloud computing hardware reliability,” in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC ’10*, 2010, p. 193.